Embedded Processor Block in Virtex-5 FPGAs

Reference Guide

UG200 (v1.8) February 24, 2010

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
01/15/08	1.0	Initial Xilinx release with ISE Design Suite 10.1.
03/31/08	1.1	• Updated to remove references to unsupported features. • Changed address `0x55` to reserved in Table 3-4. In Table 3-5 and Table 3-6, changed bits 22 and 23 to reserved. Revised "Request Priority (Level 1 arbitration)," page 43. Updated discussion on "Round-Robin Arbitration," page 45. Changed bit 23 to reserved in Table 3-37. In Table 3-41, changed bits 14–17 to reserved. In Table 3-43, changed bits 22–24 to reserved. • Changed bit 23 to reserved in Table 4-3. • Updated descriptions of bit 7 and bit 16 in Table 5-1, page 135. • Updated clock frequency ratio discussion on page 149. Updated description of CPMINTERCONNECTCLKNTO1 in Table 6-1, page 147. Updated clock frequency ratio and core reset discussions on page 149. Revised allowed CPMMCCLK ratios in Table 6-2, page 150. • In chapter 9, updated summary on page 169. Revised JTGC440TRSTNEG in Table 9-1, page 170. Added "Connecting PPC440 JTAG Logic Directly to Programmable I/O" and "Connecting PPC440 JTAG Logic in Series with the Dedicated Device JTAG Logic." • Added to "Non-Storage Instruction Execution," page 202. Updated timing diagrams and descriptions for Figure 12-13, Figure 12-14, and Figure 12-17. • Changed address `0x55` to reserved in Table 14-9. In Table 14-10 and Table 14-11, changed bits 22 and 23 to reserved. Changed bit 23 to reserved in Table 14-42. In Table 14-46, changed bits 14–17 to reserved. In Table 14-49, changed bits 22–24 to reserved.
05/07/08	1.2	• Added updated DCR addresses and added mnemonics to Table 13-5 and Table 14-52. • Revised Chapter 16, "Additional Programming Considerations," added "Bit Settings for APU/FPU Usage."
05/09/08	1.2.1	• Minor typographical change.
05/13/08	1.3	• Corrected the MI_ROWCONFLICT_MASK [0:31] register and MI_BANKCONFLICT_MASK [0:31] register on page 135 and page 269.
09/23/08	1.4	• Added index to document. • Added to the function description of C440TRCTRIGGEREVENTOUT on page 184. • Removed unsupported wildcard UDIs from page 316.

Date	Version	Revision
11/25/08	1.5	In Chapter 5, "Memory Controller Interface": • Revised "Interface Features," page 132. • In Table 5-1, page 135, revised the descriptions of Rowconflictholdenable, Bankconflictholdenable, Directionconflictholdenable, Autoholdduration, and RMW. • In Table 5-2, page 137, revised the descriptions of MIMCBANKCONFLICT and MIMCROWCONFLICT. • Updated Figure 5-4, page 139, Figure 5-5, page 140, Figure 5-6, page 141, and Figure 5-7, page 142. • Added Figure 5-8, page 142 and Figure 5-9, page 143. • Changed autostall to autohold. In Chapter 13, "DMA Controller": • Revised "DMA Operation," page 228. • Updated Figure 13-2, page 231 and Figure 13-5, page 233. • Revised second paragraph after Figure 13-2, page 231. • Added Figure 13-3, page 232, Figure 13-4, page 233, Figure 13-6, page 234, and Figure 13-7, page 235 and associated text. • Revised third paragraph after Figure 13-5, page 233. • Updated "DMA Addressing Limitation," page 236. • Rewrote "Software/Device Driver Considerations," page 240. • Corrected bit 30 description in Table 13-13, page 249. • Corrected bit 31 description in Table 13-21, page 255.
01/20/09	1.6	• Added additional information on MCMIREADDATAERR for clarification to the last paragraph of "Interface Features" on page 134. • Updated documentation references.
10/06/09	1.7	• Updated references in "Additional References," page 13. • Added third paragraph to "Locked Transfers," page 47. • Updated default value for Bit 27 in Table 4-6, page 108. • Added Table 6-3 and text describing it to "Clock Insertion Delays and PLL Usage," page 150.
02/24/10	1.8	In the Interface Timings section, page 161, added clarifying text at the end of the section relating to use of synchronous or asynchronous, master or slave DCRs.

Table of Contents

About This Guide

This reference guide is a description of the embedded processor block in Virtex®-5 FXT FPGAs. Complete and up-to-date documentation of the Virtex-5 family of FPGAs is available on the Xilinx website at http://www.xilinx.com/virtex5.

Guide Contents

This reference guide contains the following chapters:

Additional Documentation

The following documents are also available for download at http://www.xilinx.com/virtex5.

- Virtex-5 Family Overview

  The features and product selection of the Virtex-5 family are outlined in this overview.

- Virtex-5 FPGA Data Sheet: DC and Switching Characteristics

  This data sheet contains the DC and Switching Characteristic specifications for the Virtex-5 family.

- Virtex-5 FPGA User Guide

  Chapters in this guide cover the following topics:

  - Clocking Resources
  - Clock Management Technology (CMT)
  - Phase-Locked Loops (PLLs)
  - Block RAM
  - Configurable Logic Blocks (CLBs)
  - SelectIO™ Resources
  - SelectIO Logic Resources
  - Advanced SelectIO Logic Resources

- Virtex-5 FPGA RocketIO GTP Transceiver User Guide

  This guide describes the RocketIO™ GTP transceivers available in the Virtex-5 LXT and SXT platforms.

- Virtex-5 FPGA RocketIO GTX Transceiver User Guide

  This guide describes the RocketIO GTX transceivers available in the Virtex-5 TXT and FXT platforms.

- Virtex-5 FPGA Embedded Tri-Mode Ethernet MAC User Guide

  This guide describes the dedicated Tri-Mode Ethernet Media Access Controller available in the Virtex-5 LXT, SXT, TXT, and FXT platforms.

- Virtex-5 FPGA Integrated Endpoint Block User Guide for PCI Express Designs

  This guide describes the integrated Endpoint blocks in the Virtex-5 LXT, SXT, TXT, and FXT platforms used for PCI Express® designs.

- Virtex-5 FPGA XtremeDSP Design Considerations

  This guide describes the XtremeDSP™ slice and includes reference designs for using the DSP48E slice.

- Virtex-5 FPGA Configuration Guide

  This all-encompassing configuration guide includes chapters on configuration interfaces (serial and SelectMAP), bitstream encryption, Boundary-Scan and JTAG configuration, reconfiguration techniques, and readback through the SelectMAP and JTAG interfaces.

- Virtex-5 FPGA System Monitor User Guide

  The System Monitor functionality available in all the Virtex-5 devices is outlined in this guide.

- Virtex-5 FPGA Packaging and Pinout Specification

  This specification includes the tables for device/package combinations and maximum I/Os, pin definitions, pinout tables, pinout diagrams, mechanical drawings, and thermal specifications.
- Virtex-5 FPGA PCB Designer's Guide

  This guide provides information on PCB design for Virtex-5 devices, with a focus on strategies for making design decisions at the PCB and interface level.

Additional References

The following documentation provides additional information useful to this Reference Guide:

1. IBM Corp., *Book E: Enhanced PowerPC Architecture Specification*, Version 1.0, May 7, 2002
2. IBM Corp., *CoreConnect Bus Architecture Product Brief*, GK10-3116-00, September 1, 1999
3. IBM Corp., *Device Control Register Bus 3.5 Architecture Specifications*, SA14-2706-03, January 27, 2006
4. IBM Corp., *128-bit Processor Local Bus Architecture Specifications*, Version 4.6, SA-14-2538-04, July, 2004
5. IBM Corp., *PPC440x5 CPU Core User's Manual*, SA14-2613-03, Preliminary, July 15, 2003
6. Xilinx, SP006, *LocalLink Interface Specification* (v2.0), July 25, 2005 (click on the **Download** link to register for access to this specification)
7. Xilinx, DS621, *Virtex-5 FPGA Embedded Processor Block with PowerPC 440 Processor* (installed as part of the EDK)
8. Xilinx, UG018, *PowerPC 405 Processor Block Reference Guide* (does not apply to Virtex-5 FXT devices but provides useful comparisons with previous versions of Virtex devices)

Additional Support Resources

To search the database of silicon and software questions and answers, or to create a technical support case in WebCase, see the Xilinx website at: http://www.xilinx.com/support.

Typographical Conventions

This document uses the following typographical conventions. An example illustrates each convention.

Convention	Meaning or Use	Example
Italic font	References to other documents	See the *Virtex-5 FPGA Configuration Guide* for more information.
	Emphasis in text	The address (F) is asserted *after* clock event 2.
Underlined Text	Indicates a link to a web page.	http://www.xilinx.com/virtex5

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See the section "Additional Support Resources" for details. Refer to "DMA Operation" in Chapter 13 for details.
Blue, underlined text	Hyperlink to a website (URL)	Go to http://www.xilinx.com for the latest documentation.

Section I: Introduction

Chapter 1, "PowerPC 440 Embedded Processor"

PowerPC 440 Embedded Processor

Virtex®-5 FXT FPGAs introduce an embedded processor block for PowerPC® 440 processor designs. This block contains the PowerPC 440x5 32-bit embedded processor developed by IBM. The PowerPC 440x5 processor implements the IBM Book E: Enhanced PowerPC Architecture.

This chapter contains the following sections:

- "PowerPC 440 Embedded Processor Features"
- "PowerPC 440 Embedded Processor as an IBM PowerPC Implementation"
- "Processor Organization"
- "Processor Interfaces"

PowerPC 440 Embedded Processor Features

The PowerPC 440 embedded processor contains a dual-issue, superscalar, pipelined processing unit, along with other functional elements required to implement embedded system-on-a-chip solutions. These other functions include memory management, cache control, timers, and debug facilities. In addition to three separate 128-bit Processor Local Bus (PLB) interfaces, the embedded processor provides interfaces for custom coprocessors and floating-point functions, along with separate 32 KB instruction and 32 KB data caches.

The PowerPC 440 embedded processor includes the following features:

- High-performance, dual-issue, superscalar 32-bit RISC CPU
    - Superscalar implementation of the full 32-bit Book E: Enhanced PowerPC Architecture
    - Seven-stage, highly pipelined microarchitecture
    - Dual instruction fetch, decode, and out-of-order issue
    - Out-of-order dispatch, execution, and completion
    - High-accuracy dynamic branch prediction using a Branch History Table (BHT)
    - Reduced branch latency using Branch Target Address Cache (BTAC)
    - Three independent pipelines
        - Combined complex integer, system, and branch pipeline
        - Simple integer pipeline
        - Load/store pipeline
    - Single cycle multiply
    - Single-cycle multiply-accumulate (DSP instruction set extensions)
    - 9-port (6-read, 3-write) 32 x 32-bit General Purpose Register (GPR) file

    - Hardware support for all CPU misaligned accesses
    - Full support for both big- and little-endian byte ordering
    - Power management features
- Primary caches
    - 32 KB instruction cache
    - 32 KB data cache
    - Single-cycle access
    - 32-byte (eight word) line size
    - 64-way associativity
    - Write-back and write-through operation
    - Control over whether stores allocate or write-through on cache miss
    - Extensive load/store queues and multiple line fill/flush buffers
    - Non-blocking with up to four outstanding load misses
    - Cache line locking supported
    - Caches can be partitioned to provide separate regions for "transient" instructions and data
        - High associativity permits efficient allocation of cache memory
    - Critical word first data access and forwarding
    - Cache tags and data are parity-protected against soft errors
- Memory Management Unit (MMU)
    - Separate instruction and data shadow TLBs
    - 64-entry, fully associative unified TLB array
    - Variable page sizes (1 KB - 256 MB), simultaneously resident in TLB
    - MMU supports 4-bit extended address bits (can formulate a 36-bit real address)
    - Flexible TLB management with software page table search
    - Storage attribute controls for write-through, caching inhibited, guarded, and byte order (endianness)
    - Four user-definable storage attribute controls (for controlling CodePack™ code compression and transient data, for example)
    - TLB tags and data are parity-protected against soft errors
- Debug facilities
    - Extensive hardware debug facilities incorporated into the IEEE 1149.1 JTAG port
        - Multiple instruction and data address breakpoints (including range)
        - Data value compare
        - Single-step, branch, trap, and other debug events
    - Non-invasive, real-time software trace interface
- Timer facilities
    - 64-bit time base
    - Decrementer with auto-reload capability
    - Fixed Interval Timer (FIT)
    - Watchdog timer with critical interrupt and/or auto-reset

- Multiple embedded processor interfaces defined by the IBM CoreConnect on-chip system architecture
  - PLB interfaces
    - Three independent 128-bit interfaces (internal to the embedded processor block in Virtex-5 FPGAs) for instruction reads, data reads, and data writes
    - Multiple CPU:PLB frequency ratios supported
  - Auxiliary Processor Unit (APU) Port
    - Functional extensions provided to the processor pipelines, including GPR file operations
    - 128-bit load/store interface (direct access between the APU and the primary data cache)
    - Interface can support APU execution of all PowerPC floating-point instructions
    - Attachment capability for DSP coprocessing such as accumulators and SIMD computation
    - Enables customer-specific instruction enhancements
  - Device Control Register (DCR) interface for independent access to on-chip control registers
    - Avoids contention for high-bandwidth PLB system bus
  - Clock and power management interface
  - JTAG debug interface

PowerPC 440 Embedded Processor as an IBM PowerPC Implementation

The PowerPC 440 embedded processor implements the full, 32-bit fixed-point subset of the IBM Book E: Enhanced PowerPC architecture. The PowerPC 440 embedded processor fully complies with this architectural specification. The 64-bit operations of the architecture are not supported, and the embedded processor does not implement the floating-point operations, although a floating-point unit (FPU) can be attached (using the APU interface). Within the embedded processor, the 64-bit operations and the floating-point operations are trapped, and the floating-point operations can be emulated using software.

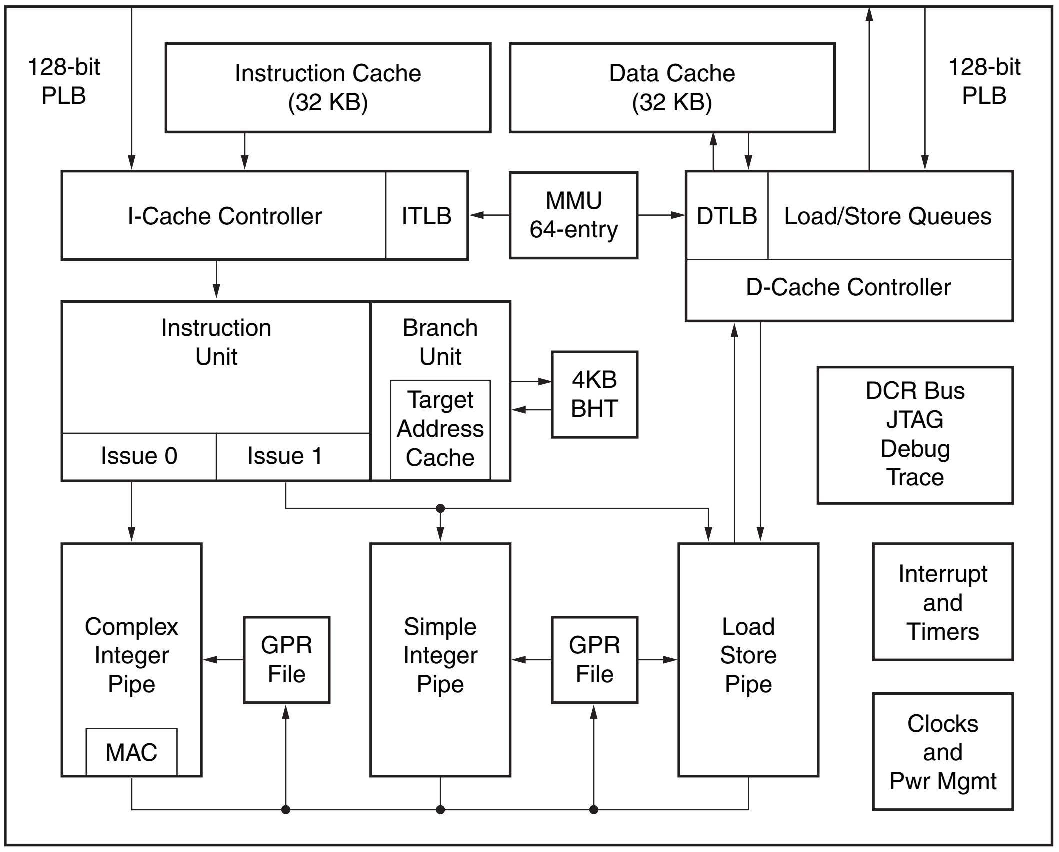
See Appendix A, "Guidelines for 32-bit Book E" in *Book E: Enhanced PowerPC Architecture Specification* [Ref 1] for more information on 32-bit subset implementations of the architecture.

Note: This document differs from the Book E architecture specification in the use of bit numbering for architected registers. Specifically, Book E defines the full, 64-bit instruction set architecture, where all registers have bit numbers from 0 to 63, with bit 63 being the least significant. This document describes the PowerPC 440 embedded processor, which is a 32-bit subset implementation of the architecture. Accordingly, all architected registers are 32 bits in length, with the bits numbered from 0 to 31, where bit 31 is the least significant. Therefore, references to register bit numbers from 0 to 31 in this document correspond to bits 32 to 63 of the same register in the Book E architecture specification.

Processor Organization

The PowerPC 440 embedded processor includes a seven-stage pipelined PowerPC processor, which consists of a three-stage, dual- issue instruction fetch and decode unit with attached branch unit, together with three independent, four-stage pipelines for complex integer, simple integer, and load/store operations, respectively. The PowerPC 440 embedded processor also includes a memory management unit (MMU), separate instruction and data cache units, JTAG, debug, and trace logic, and timer facilities.

Figure 1-1 illustrates the logical organization of the PowerPC 440 embedded processor.



UG200_c1_01_022707

Figure 1-1: **Block Diagram of PowerPC 440 Embedded Processor**

Superscalar Instruction Unit

The instruction unit of the PowerPC 440 embedded processor fetches, decodes, and issues two instructions per cycle to any combination of the three execution pipelines and/or the APU interface. The instruction unit includes a branch unit, which provides dynamic branch prediction using a branch history table (BHT), as well as a branch target address cache (BTAC). These mechanisms greatly improve the branch prediction accuracy and reduce the latency of taken branches, such that the target of a branch can usually be executed immediately after the branch itself, with no penalty.

Execution Pipelines

The PowerPC 440 embedded processor contains three execution pipelines: complex integer, simple integer, and load/store. Each pipeline consists of four stages and can access the nine-ported (six read, three write) GPR file. There are two identical copies of the GPR file to improve performance and avoid contention for it. One copy is dedicated to the complex integer pipeline, while the other is shared by the simple integer and the load/store pipelines.

The complex integer pipeline handles all arithmetic, logical, branch, and system management instructions (such as interrupt and TLB management, move to/from system registers, and so on). This pipeline also handles multiply and divide operations, and 24 DSP instructions that perform a variety of multiply-accumulate operations. The complex integer pipeline multiply unit can perform 32-bit x 32-bit multiply operations with single-cycle throughput and three-cycle latency; 16-bit x 32-bit multiply operations have only two-cycle latency. Divide operations take 33 cycles.

The simple integer pipeline can handle most arithmetic and logical operations, which do not update the Condition Register (CR).

The load/store pipeline handles all load, store, and cache management instructions. All misaligned operations are handled in hardware with no penalty on any operation contained within an aligned 16-byte region. The load/store pipeline supports all operations to both big-endian and little-endian data regions.

Instruction and Data Cache Controllers

The PowerPC 440 embedded processor provides separate instruction and data cache controllers and 32 KB arrays, which allow concurrent access and minimize pipeline stalls. Both cache controllers have 32-byte lines, and both are 64-way set-associative. Both caches support parity checking on the tags and data in the memory arrays to protect against soft errors. If a parity error is detected, the CPU causes a machine check exception.

The PowerPC instruction set provides a rich set of cache management instructions for software-enforced coherency. The PowerPC 440 implementation also provides special debug instructions that can directly read the tag and data arrays.

The instruction cache controller connects to the instruction-side PLB interface of the processor. The data cache controller connects to the data read and data write PLB interfaces.

Instruction Cache Controller (ICC)

The ICC delivers two instructions per cycle to the instruction unit of the PowerPC 440 embedded processor. The ICC also handles the execution of the PowerPC instruction cache management instructions for coherency. The ICC includes a speculative prefetch mechanism. These speculative pre-fetches can be abandoned if the instruction execution branches away from the original instruction stream.

Note: Speculative prefetching should *not* be used with this version of the PowerPC 440 processor because of known errors documented by IBM.

The ICC supports cache line locking at 16-line granularity. In addition, the notion of a "transient" portion of the cache is supported, in which the cache can be configured such that only a limited portion is used for instruction cache lines from memory pages designated by a storage attribute from the MMU as being transient in nature. Such memory pages would contain code that is unlikely to be reused once the processor moves on to the next series of instruction lines. Thus performance may be improved by preventing each series of instruction lines from overwriting all of the “regular” code in the instruction cache.

Data Cache Controller (DCC)

The DCC handles all load and store data accesses, as well as the PowerPC data cache management instructions. All misaligned accesses are handled in hardware. Those accesses contained within a halfline (16 bytes) are handled as a single request. Load and store accesses that cross a 16-byte boundary are broken into two separate accesses by the hardware.

The DCC interfaces to the APU port to provide direct load/store access to the data cache for APU load and store operations. Such APU load and store instructions can access up to 16 bytes (one quadword) in a single cycle.

The data cache can be operated in a store-in (copy-back) or write-through manner, according to the write-through storage attribute specified for the memory page by the MMU. The DCC also supports both *store-with-allocate* and *store-without-allocate* operations, such that store operations that miss in the data cache can either “allocate” the line in the cache by reading it in and storing the new data into the cache, or alternatively bypass the cache on a miss and simply store the data to memory. This characteristic can also be specified on a page-by-page basis by a storage attribute in the MMU.

The DCC also supports cache line locking and “transient” data in the same manner as the ICC (as described in “Instruction Cache Controller (ICC)”).

The DCC provides extensive load, store, and flush queues, such that up to three outstanding line fills and up to four outstanding load misses can be pending, and the DCC can continue servicing subsequent load and store hits in an out-of-order fashion. Store-gathering can also be performed on caching inhibited, write-through, and without-allocate store operations for up to 16 contiguous bytes. Finally, each cache line has four separate dirty bits (one per doubleword), so that the amount of data flushed on cache line replacement can be minimized.

Memory Management Unit (MMU)

The PowerPC 440 MMU generates a 36-bit real address as part of the translation process from the 32-bit effective address, which is calculated by the processor as an instruction fetch or load/store address. However, only a 32-bit (4 GB) address space is accessible in Xilinx EDK systems. The high-order 4 bits of the 36-bit real address must be all zeros.

The MMU provides address translation, access protection, and storage attribute control for embedded applications. The MMU supports demand paged virtual memory and other management schemes that require precise control of logical to physical address mapping and flexible memory protection. Working with appropriate system-level software, the MMU provides the following functions:

- Translation of the 32-bit effective address space into the 36-bit real address space
- Page level read, write, and execute access control
- Storage attributes for cache policy, byte order (endianness), and speculative memory access
- Software control of page replacement strategy

The translation lookaside buffer (TLB) is the primary hardware resource involved in the control of translation, protection, and storage attributes. It consists of 64 entries, each specifying the various attributes of a given page of the address space. The TLB is fully associative; the entry for a given page can be placed anywhere in the TLB. The TLB tag and data memory arrays are parity protected against soft errors. If a parity error is detected, the CPU causes a machine check exception.

Software manages the establishment and replacement of TLB entries, which gives system software significant flexibility in implementing a custom page replacement strategy. For example, to reduce TLB thrashing or translation delays, software can reserve several TLB entries for globally accessible static mappings. The instruction set provides several instructions for managing TLB entries. These instructions are privileged and the processor must be in supervisor state for them to be executed.

The first step in the address translation process is to expand the effective address into a virtual address. The 32-bit effective address is appended to an 8-bit Process ID (PID) as well as a 1-bit "address space" identifier (AS). The PID value is provided by the PID register. The AS identifier is provided by the Machine State Register (MSR), which contains separate bits for the instruction fetch address space (MSR[IS]) and the data access address space (MSR[DS]). Together, the 32-bit effective address, the 8-bit PID, and the 1-bit AS form a 41-bit virtual address. This 41-bit virtual address is then translated into the 36-bit real address using the TLB.

The MMU divides the address space (effective, virtual, or real) into pages. Eight page sizes (1 KB, 4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 16 MB, 256 MB) are simultaneously supported, such that at any given time the TLB can contain entries for any combination of page sizes. For an address translation to occur, a valid entry for the page containing the virtual address must be in the TLB. An attempt to access an address for which no TLB entry exists causes an Instruction (for fetches) or Data (for load/store accesses) TLB Error exception.

To improve performance, both the instruction cache and the data cache maintain separate *shadow* TLBs. The instruction shadow TLB (ITLB) contains four entries, while the data shadow TLB (DTLB) contains eight. These shadow arrays minimize TLB contention between instruction fetch and data load/store operations. The instruction fetch and data access mechanisms only access the main 64-entry unified TLB when a miss occurs in the respective shadow TLB. The penalty for a miss in either of the shadow TLBs is three cycles. Hardware manages the replacement and invalidation of both the ITLB and DTLB. No system software action is required.

Each TLB entry provides separate user state and supervisor state read, write, and execute permission controls for the memory page associated with the entry. If software attempts to access a page for which it does not have the necessary permission, an Instruction (for fetches) or Data (for load/store accesses) Storage exception occurs.

Each TLB entry also provides a collection of storage attributes for the associated page. These attributes control cache policy (such as cachability and write-through as opposed to copy-back behavior), byte order (big endian as opposed to little endian), and enabling of speculative access for the page. In addition, a set of four, user-definable storage attributes is provided. These attributes can be used to control various system-level behaviors, such as instruction compression using IBM CodePack technology. They can also be configured to control whether data cache lines are allocated upon a store miss, and whether accesses to a given page should use the *normal* or *transient* portions of the instruction or data cache.

More details on the MMU implementation and the MMU programming model are available in the *PPC440x5 CPU Core User's Manual* [Ref 5].

Timers

The PowerPC 440 embedded processor contains a time base and three timers: a decrementer (DEC), a fixed interval timer (FIT), and a Watchdog Timer. The time base is a 64-bit counter that gets incremented at a frequency either equal to the processor clock rate or as controlled by a separate asynchronous timer clock input to the embedded processor. No interrupt is generated as a result of the time base wrapping back to zero.

The DEC is a 32-bit register that is decremented at the same rate at which the time base is incremented. The user loads the DEC register with a value to create the desired interval. When the register is decremented to zero, a number of actions occur: the DEC stops decrementing, a status bit is set in the Timer Status register (TSR), and a decrementer exception is reported to the interrupt mechanism of the PowerPC 440 embedded processor. Optionally, the DEC can be programmed to automatically reload the value contained in the Decrementer Auto-Reload register (DECAR), after which the DEC resumes decrementing. The Timer Control register (TCR) contains the interrupt enable for the decrementer interrupt.

The FIT generates periodic interrupts based on the transition of a selected bit from the time base. Users can select one of four intervals for the FIT period by setting a control field in the TCR to select the appropriate bit from the time base. When the selected time base bit transitions from 0 to 1, a status bit is set in the TSR, and a Fixed Interval Timer exception is reported to the interrupt mechanism of the PowerPC 440 embedded processor. The FIT interrupt enable is contained in the TCR.

Similar to the FIT, the watchdog timer also generates a periodic interrupt based on the transition of a selected bit from the time base. Users can select one of four intervals for the watchdog period, again by setting a control field in the TCR to select the appropriate bit from the time base. Upon the first transition from 0 to 1 of the selected time base bit, a status bit is set in the TSR, and a watchdog timer exception is reported to the interrupt mechanism of the PowerPC 440 embedded processor. The watchdog timer can also be configured to initiate a hardware reset if a second transition of the selected time base bit occurs prior to the first watchdog exception being serviced. This capability provides an extra measure of recoverability from potential system lock-ups.

Debug Facilities

The PowerPC 440 debug facilities include debug modes for the various types of debugging used during hardware and software development. Also included are debug events that allow developers to control the debug process. Debug modes and debug events are controlled using debug registers in the embedded processor. The debug registers are accessed either through software running on the processor or through the JTAG port.

The next subsection provides a brief overview of the debug modes and development tool support. More details on the debug control registers and their programming are available in the *PPC440x5 CPU Core User's Manual* [Ref 5].

Debug Modes

The PowerPC 440 embedded processor supports four debug modes: internal, external, real-time trace, and debug wait. Each mode supports a different type of debug tool used in embedded systems development. Internal debug mode supports software-based ROM monitors, and external debug mode supports a hardware emulator type of debug. Real-time trace mode uses the debug facilities to indicate events within a trace of processor execution in real time. Debug wait mode enables the processor to continue to service real-time critical interrupts while instruction execution is otherwise stopped for hardware debug. The debug modes are controlled by Debug Control Register 0 (DBCR0) and the setting of bits in the Machine State Register (MSR).

Internal debug mode supports accessing architected processor resources, setting hardware and software breakpoints, and monitoring processor status. In internal debug mode, debug events can generate debug exceptions, which can interrupt normal program flow so that monitor software can collect processor status and alter processor resources.

Internal debug mode relies on exception-handling software—running on the processor—along with an external communications path to debug software problems. This mode is used while the processor continues executing instructions and enables debugging of problems in application or operating system code. Access to debugger software executing in the processor while in internal debug mode can be established through a communications port in the system, such as a serial port or Ethernet connection.

External debug mode supports stopping, starting, and single-stepping the processor, accessing architected processor resources, setting hardware and software breakpoints, and monitoring processor status. In external debug mode, debug events can architecturally "freeze" the processor. While the processor is frozen, normal instruction execution stops, and the architected processor resources can be accessed and altered using a debug tool attached through the JTAG port. This mode is useful for debugging hardware and low-level control software problems.

Processor Interfaces

The interfaces to the PowerPC 440 embedded processor include:

- Processor Local Bus (PLB)
- Device configuration register (DCR) interface
- Auxiliary processor unit (APU) port
- JTAG, debug, and trace ports
- Interrupt interface
- Clock and power management interface

Some of these interfaces are described briefly in the following subsections.

Processor Local Bus (PLB)

There are three independent 128-bit PLB interfaces to the PowerPC 440 embedded processor. One PLB interface supports instruction cache reads, while the other two support data cache reads and writes. All three PLB interfaces are connected as masters to the crossbar in the embedded processor block in Virtex-5 FPGAs.

The data cache PLB interfaces make requests for 32-byte lines, as well as for 1 to 15 bytes within a 16-byte (quadword) aligned region. A 16-byte line request is used for quadword APU load operations to caching inhibited pages, and for quadword APU store operations to caching inhibited, write-through, or without allocate pages.

The instruction cache controller makes 32-byte line read requests.

Each of the PLB interfaces fully supports the address pipelining capabilities of the PLB, and in fact can go beyond the pipeline depth and minimum latency that the PLB supports. Specifically, each interface supports up to three pipelined request/acknowledge sequences prior to performing the data transfers associated with the first request. For the data cache, if each request must be broken into three separate transactions (for example, for a misaligned doubleword request to a 32-bit PLB slave), then the interface actually supports up to nine outstanding request/acknowledge sequences prior to the first data transfer. Furthermore, each PLB interface tolerates a zero-cycle latency between the request and the address and data acknowledge (that is, the request, address acknowledge, and data acknowledge may all occur in the same cycle).

The PLB interfaces described above are not directly visible to the Virtex-5 FXT FPGA user. These interfaces are connected to the crossbar described in Chapter 3, "Crossbar." The Virtex-5 FXT FPGA user sees only the external interfaces on the embedded processor block, which includes the PowerPC 440 and the crossbar interfaces. These external interfaces are described in Chapter 2, "Embedded Processor Block Overview," and the subsequent chapters.

Device Control Register (DCR) Interface

The DCR interface provides a mechanism for the PowerPC 440 embedded processor to set up and check status of other hardware facilities in the embedded processor block in the Virtex-5 FPGA and elsewhere in the system. DCRs are accessed through the PowerPC **`mfdcr`** and **`mtdcr`** instructions.

The interface is interlocked with control signals such that it can be connected to peripheral units that can be clocked at different frequencies from the embedded processor.

The DCR interface also allows the PowerPC 440 embedded processor to communicate with peripheral devices without using the PLB interface, avoiding the impact to the primary system bus bandwidth, and without additional segmentation of the usable address map.

Auxiliary Processor Unit (APU) Port

This interface provides the PowerPC 440 embedded processor with the flexibility for attaching a tightly coupled, coprocessor-type macro incorporating instructions that go beyond those provided within the embedded processor itself. The APU port provides sufficient functionality for attachment of various coprocessor functions, such as a fully compliant PowerPC floating-point unit, or other custom function implementing algorithms appropriate for specific system applications. The APU interface supports dual-issue pipeline designs, and can be used with macros that contain their own register files, or with simpler macros that use the CPU GPR file for source and/or target operands. APU load and store instructions can directly access the PowerPC 440 data cache with operands of up to a quadword (16 bytes) in length.

The APU interface provides the capability for a coprocessor to execute concurrently with the PowerPC 440 embedded processor instructions that are not part of the PowerPC instruction set. Accordingly, areas have been reserved within the architected instruction space to allow for these customer-specific or application-specific APU instruction set extensions.

JTAG Port

The JTAG port is enhanced to support the attachment of a debug tool. Through the JTAG test access port, and using the debug facilities designed into the PowerPC 440 embedded processor, a debug tool can single-step the processor and interrogate internal processor state to facilitate hardware and software debugging. The enhancements, which comply with the IEEE 1149.1 specification for vendor-specific extensions, are therefore compatible with standard JTAG hardware for Boundary-Scan system testing.

Section II: Embedded Processor Block

Chapter 2

Embedded Processor Block Overview

The embedded processor block in Virtex-5 FXT devices contains several additional modules along with the PowerPC 440 processor. These additional modules allow system designers to improve the performance and reduce the cost of their designs. This chapter provides an overview of the embedded processor block in Virtex-5 FPGAs and briefly describes each of the additional modules and interfaces.

Embedded Processor Block Components

The main components of the embedded processor block in Virtex-5 FXT FPGAs are the processor, the crossbar and its interfaces, the Auxiliary Processing Unit (APU) controller, and the control (clock and reset) module. Figure 2-1 shows the embedded processor block and its components.

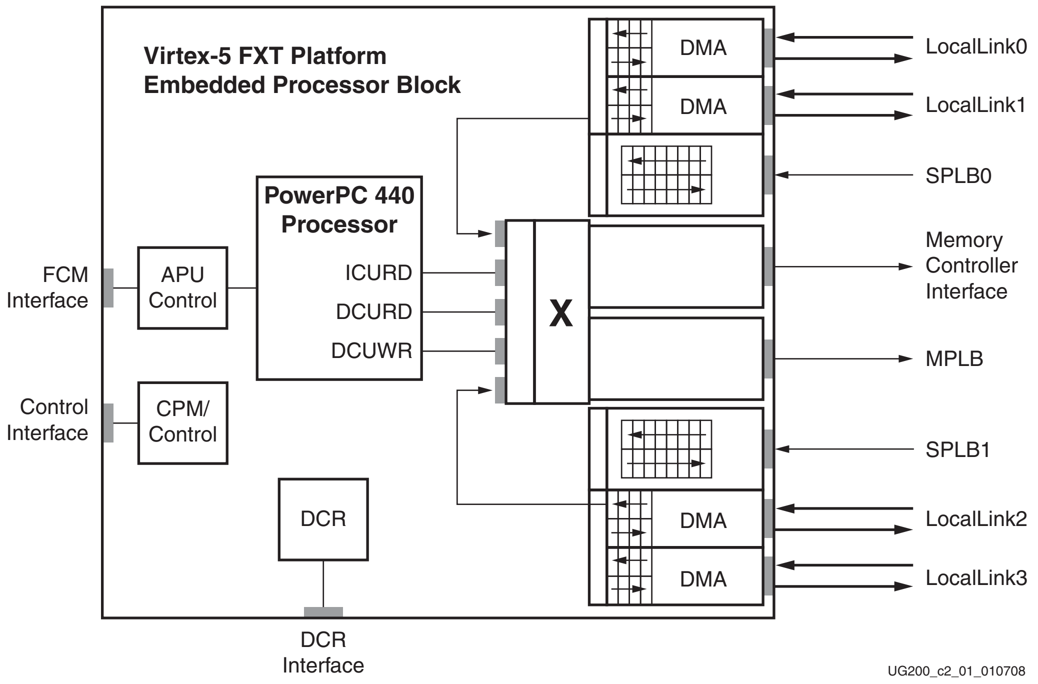


Figure 2-1: **Embedded Processor Block in Virtex-5 FPGAs**

The processor is described in detail in Chapter 1, "PowerPC 440 Embedded Processor." The processor has three PLB interfaces: one for instruction reads, one for data reads, and one for data writes. Typically, all three interfaces access a single large external memory. Peripheral access in PowerPC 440 systems is memory mapped, and the data PLB interfaces typically connect to various peripherals directly or via bridges. Some of these peripherals might have Direct Memory Access (DMA) capability to improve data bandwidth and performance. Other peripherals might rely on a separate DMA engine to provide this improved data bandwidth between the peripheral and memory. Peripherals can be implemented in soft logic, using the lookup tables (LUTs) and other primitive logic elements provided by the FPGA, or the peripherals can be implemented in silicon. Peripherals are hardened or implemented in silicon if they are likely to be used by a large number of customers, or if hardening is necessary for performance reasons. Some peripherals are implemented in Virtex-5 FXT silicon, such as integrated endpoints for PCI Express designs and tri-mode Ethernet MACs implemented in silicon. These peripherals have a LocalLink interface for high-bandwidth data transfers.

Crossbar and its Interfaces

The crossbar and its interfaces allow the processor with its three PLB interfaces, soft peripherals with PLB interfaces, and peripherals with LocalLink interfaces to share access to a high-performance memory controller. As shown in Figure 2-1, the crossbar has:

- Five PLB slave interfaces
  - Three for the PLB interfaces from the processor
  - Two for soft peripherals with PLB interfaces to allow these peripherals to access the high-speed memory controller interface
- Four full-duplex LocalLink channels with built-in DMA control and access to the memory controller interface
- One high-speed memory controller interface that hardens several parts of a typical memory controller but leaves the physical interface to the memory to be implemented as soft logic for reasons of flexibility
- One PLB master interface to allow the processor to connect to other peripherals in the FPGA logic

Details of the crossbar capabilities are documented in Chapter 3, "Crossbar," and details of the crossbar interfaces that interface to the Virtex-5 FPGA logic are documented in Chapter 4, "PLB Interface," and Chapter 5, "Memory Controller Interface."

Control and other Interfaces

The embedded processor block and the processor have several other standard interfaces. The clock, power management, and reset interfaces are described in more detail in Chapter 6, "Reset, Clock, and Power Management Interfaces."

The processor has a Device Control Register (DCR) interface that allows control registers of peripherals to be connected to a DCR bus and accessed through the register space of the processor. The processor block has an additional DCR slave interface that allows external peripherals to act as DCR masters and access the registers on the hardened DMA controllers within the processor block. The DCR interface is documented in Chapter 7, "Device Control Register Bus." The Interrupt interface of the processor is documented in Chapter 8, "Interrupt Controller Interface," while the JTAG interface is documented in Chapter 9, "JTAG Interface." The debug and trace interfaces are documented in Chapter 10, "Debug Interface," and Chapter 11, "Trace Interface," respectively.

Auxiliary Processor Unit Controller

The embedded processor block in Virtex-5 FPGAs includes a hardened Auxiliary Processor Unit (APU) controller driven by the APU interface on the processor. The APU interface on the processor allows users to build an auxiliary processor to execute instructions that are not part of the PowerPC 440 instruction set. However, this interface requires the auxiliary processor to be clocked at the CPU speed and also be in complete lock-step with the processor pipeline. The processor can run much faster than a soft core implemented on the FPGA logic, so an auxiliary processor implemented in soft logic would force the processor to run at a lower speed, reducing the performance gain. The APU controller directs and synchronizes the CPU pipeline, allowing the soft auxiliary processor and the CPU to run at different clock rates. Additionally, the APU controller can decode the instructions on behalf of the soft auxiliary processor unit, resulting in faster overall instruction execution for the instructions using the auxiliary processor. The APU controller and its interface to the FPGA logic are described in detail in Chapter 12, "Auxiliary Processor Unit Controller."

Direct Memory Access Controller

The processor block includes a hardened Direct Memory Access (DMA) controller that allows peripherals to directly transfer data to and from a memory controller connected to the processor block via the memory controller interface or the PLB interface. The DMA controller can be monitored and controlled through its Device Control Registers (DCRs). The DMA controller has LocalLink data interfaces to peripherals. More information on the DMA controller and its interfaces is available in Chapter 13, "DMA Controller."

Chapter 3

Crossbar

Overview

The crossbar acts as a central arbitration and switching module that accepts master requests from up to five groups of master devices and redirects the transactions to one of two groups of slave devices. The crossbar also directs the responses from the slave devices back to the correct master devices. All data passing from any master device to any slave device within the embedded processor block in Virtex-5 FPGAs passes through the crossbar.

Along with the processor, the crossbar is a hard block instantiated in silicon within the Virtex-5 FPGA family. The crossbar forms the main interface into or out of the CPU. The crossbar is also the main connection and switch point for any devices instantiated within the FPGA logic that need to communicate with the processor or external memory visible to the processor.

The crossbar functions conceptually as a simple switch. If a master asks for access to a slave and wins arbitration, the crossbar acts as a switch to connect the requesting master with the requested slave. This topology allows for a high-speed interconnect with an efficient linkage of many high-performance masters. However, unlike a bus-only based topology, transactions from one master to a slave are not always visible to all masters. This might violate some bus-based ordering assumptions, which are discussed in"Usage Notes and Limitations," page 55.

Note: To simplify discussions, the term *Crossbar* in this document is defined as a block that consists of internal modules that provide both switching and bridging functions.

Figure 2-1, page 29 shows the crossbar and its interfaces.

The crossbar has the following interfaces:

- ICURD: Instruction Cache Unit Read PLB interface of the processor
- DCUWR: Data Cache Unit Write PLB interface of the processor
- DCURD: Data Cache Unit Read PLB interface of the processor
- SPLB 0: Slave PLB 0 port used to attach soft PLB masters implemented in FPGA logic to the embedded processor block
- SPLB 1: Slave PLB 1 port used to attach soft PLB masters implemented in FPGA logic to the embedded processor block
- MPLB: Master PLB interface used to attach slaves implemented in FPGA logic to the embedded processor block
- MCI: Memory controller interface used to attach high data rate memory controllers to the embedded processor block. The MCI provides a simple protocol that allows the soft memory controller to run at higher speeds because it does not need to implement the more complex and more general PLB protocol. The MCI also decouples the high

data rate memory from the MPLB slave interface, which thus can be used for slow peripherals.

- LocalLink/DMA: Four LocalLink interfaces to the internal DMA engines and the MPLB and MCI interfaces. Two LocalLink interfaces and an SPLB interface are locally arbitrated before being arbitrated with the processor PLB interfaces for accessing the MPLB and MCI, as shown in Figure 2-1, page 29.

With the exception of the LocalLink and MCI interfaces, all of the interfaces are Processor Local Bus (PLB), adhering to the *PLB Architecture Specification* [Ref 4]. The MCI forms the main interface to and from memory for both the processor as well as any of the PLB devices.

Note: Ports or interfaces on the crossbar are named as per their roles on the PLB to which they attach. For example, the SPLB 1 port on the crossbar acts as a PLB slave on the PLB that can be attached to that port. If a master device on a PLB connected to the SPLB 1 crossbar port wishes to transact with a slave device connected through the crossbar's MPLB port, the connection is the PLB master connects through the PLB to the crossbar SPLB port, which connects through the crossbar to the MPLB port, connecting through the PLB to the slave device.

The MPLB, SPLB 1, and SPLB 0 ports can be connected to PLB system buses with any desired mix of other PLB master and slave devices.

The crossbar sits between the five slave interfaces in the embedded processor block (three slave interfaces to the processor and one each from the two SPLBs) and the two master interfaces (the MPLB and the MCI). The crossbar redirects requests coming from the slave interfaces to either master interface based on the address map and also redirects the data phase from the slave to the respective requesting master. If multiple master requests occur concurrently, the crossbar arbitrates between the multiple masters, lets the highest priority master through, and buffers the other master(s) request.

The crossbar's PLB interface adheres to the *PLB Architecture Specification* [Ref 4]. The address width is 36 bits with 128-bit data. The embedded processor block in Virtex-5 FPGAs supports 36-bit physical addressing, but the top four bits are defined as zeros within the Embedded Development Kit (EDK) tools and IP. PLB rules for larger buses configuring to smaller bus widths are adhered to. Data transfer between slave ports (SPLB 0 and SPLB 1) is not supported.

Key Features

The key features of the crossbar are as follows:

- The crossbar provides command pipelining for up to five transactions to the MCI and MPLB, which can be running concurrently in the embedded processor block in Virtex-5 FXT FPGAs. This allows the latency of the address phase to be hidden by overlapping it with the data phase.
- Supported commands are single cycle, line transfers, fixed length burst, and indeterminate length burst.
- Independent arbitration: one arbiter for the MPLB and another for the MCI. Transactions can be issued to both the MPLB and the MCI at the same time.
- Address switch function: receives the primary and secondary commands from the masters and issues the command to the correct slave.
- Data switch function: requires switching the correct master and slave buses together, which enables the transaction, detects the end of the transaction, and switches to the next data phase with minimal latency.
- The command and data phases are independent.

- The arbiter is required to remember the order and context (master and slave) of each command so that the data phases are executed in the same order.
- Read and write data phases are independent and can run concurrently. There is a separate command FIFO for each direction.
- Multiple crossbar to FPGA logic clock ratios are supported to optimize performance.
- All internal crossbar transactions are 128 bits wide. PLB rules that allow the use of 64-and 32-bit wide devices are supported by the crossbar block.
- The overall address space for the system is 64 Gbytes (36 bits of address) that can be divided between the slaves. Various memory mappings can be configured through bitstream or DCR operations.

Note: The embedded processor block in Virtex-5 FXT FPGAs supports 36-bit physical addressing, but the top four bits are defined as zeros within the Embedded Development Kit (EDK) tools and IP. Thus the available address space is 4 GB.

Hardware Description

Overview

The crossbar interconnects the processor and two SPLB interfaces to the memory interface and the MPLB interface. This function allows for any PLB master in the device (including the processor) to read and write to any memory-mapped location in the device connected to the crossbar. It supports a 128-bit data bus, a 36-bit address bus, and memory mapping to determine whether the memory interface or the MPLB is selected for the transfer. Most optional features in the PLB specification are supported, including bus lock, rearbitration, slave wait, master abort, master or slave terminate, and TAttribute.

The transfer types supported are single word, cache line, fixed length burst, and indeterminate burst. Transfers can be unaligned for single word transfers. Also, word steering, mirroring, conversion cycle, and burst length adjustment are handled automatically. Thus, mismatched bus sizes are allowed. Any combination of 32-, 64-, and 128-bit masters can transact with any 32-, 64-, or 128-bit slave. All burst widths (from word to quadword) are supported.

The crossbar consists of two identical pieces: an arbiter and switch for the MPLB and another for the memory interface. If multiple masters request at the same cycle, the crossbar arbitrates between the multiple masters and asserts an acknowledge to the winning master. The arbiter then decides the order of execution and places each command into the command queue. Crossbar arbitration is a two-step process that consists of a request and master priority levels. Available arbitration algorithms are fixed priority, least recently used, and round robin.

The output command queue contains commands that are waiting for the data phase to begin. This queue saves the context for the command so that the correct master can be connected when the data phase begins. There are two output command queues: one for reads and one for writes, which allows for concurrent read and write transfers for full utilization of the PLB read and write channels. See "Ordering Requirement of Transactions in the Crossbar," page 49 for further details on read and write ordering.

When a command reaches the front of the queue and the previous transfer is complete, the data phase controller selects the correct master (via the muxes and demuxes) to connect to and enables the transfer. If there are no commands in the output queue, the command issuer instead initiates the data phase.

At the end of the data transfer, the data phase controller removes the current command from the queue and starts the data phase for the next command without dead cycles between the data phases.

The primary function of the PLB interfaces is to adapt the crossbar transaction rate to the transaction rates of the soft masters and slaves. There are three causes for the rate mismatch:

- Clock frequency

  In typical applications, the crossbar runs at a higher clock frequency than that of the soft masters and slaves, resulting in a higher raw throughput for the crossbar. This is always the case when comparing the crossbar with the MPLB or the MCI because the crossbar contains two almost independent arbiters that separately drive the MPLB and the MCI. The situation is more complex when comparing the crossbar and the SPLB, as discussed below.

- Arbitration success

  The arbitration process in the crossbar divides the available crossbar bandwidth among the various active PLB masters that are directly connected to the crossbar. So an SPLB that sends and receives data to and from the MPLB only gets a fraction of the crossbar bandwidth if the processor and/or the other SPLB are also accessing the MPLB. (It is a similar situation for an SPLB sending or receiving to or from the MCI.) Because there are three PLB write masters (the instruction cache does not have a write master) and four read masters connected to the crossbar with a fair arbitration scheme, the SPLB gets, on average, at least one-third of the crossbar write bandwidth or the full SPLB write bandwidth (whichever is less) and at least one-fourth of the average crossbar read bandwidth or the full SPLB read bandwidth (whichever is less). If a master does not request crossbar access, that time slot is available for another master to use.

- Bus width conversion

  The PLB interfaces and the crossbar are 128-bit devices, and the interfaces support 128-bit soft devices natively. In addition, the interfaces also support 32-bit and 64-bit PLB devices at the expense of a reduced throughput. 32-bit and 64-bit soft masters and slaves have throughputs of one quarter and one half, respectively, of that of a corresponding 128-bit device.

The three rate mismatch factors combine dynamically together and can result in a mismatched transaction rate between devices at any given instant in time. Rate adaptation, which is needed to avoid possible data loss, is achieved through the built-in flow control mechanism of the PLB protocol together with the use of FIFOs for command and data buffering to absorb temporary differences in transaction rates.

Another major function of the PLB interfaces, as mentioned in the bus width conversion bullet, is to convert transactions from 32-bit and 64-bit soft masters and slaves into 128-bit format, and vice versa. The conversion process involves command modification, data mirroring, steering, and packing(1). Packing of burst transfers is not an inherent requirement for conversion but is done to improve the crossbar bandwidth efficiency, and because the MCI can only handle packed data.

In addition, each SPLB also contains arbitration logic to allow the sharing of the crossbar access with two 32-bit DMA controllers. Each SPLB and the two corresponding DMA controllers share the same crossbar interface logic located in the SPLB.

1. Packing is done for burst transfers and line transfers. Single transfers are not packed.

By default, a simple round-robin priority scheme is used to arbitrate among the SPLB and LocalLink/DMA channels with an initial ordering of SPLB0-DMA0-DMA1 and SPLB1-DMA2-DMA3. The priority of each DMA channel can be increased by setting the DMA priority bits in the SPLB Configuration registers as described in "Device Control Registers (DCRs)," page 56. When more than one interface has the same priority for this local arbitration, a round-robin scheme is used for the interfaces that have the same priority with the same ordering as described earlier.

Note: If the LocalLink/DMA interfaces are used but the corresponding SPLB interface is not connected, the clock pin of that SPLB interface must still be connected to a valid clock signal.

Hardware Interface

All interfaces at the boundary of the crossbar have 128-bit data buses, which simplifies the logic required to be implemented within the embedded processor block. Accesses from FPGA logic masters and slaves that do not have 128-bit data buses are converted to 128-bit accesses internally. All accesses through the crossbar are fixed length bursts and cannot be terminated except for bursts performed by the processor's ICURD interface.

Slave Ports

Instruction Side (ICURD)

The processor uses its instruction-side read (ICURD) PLB interface to perform instruction reads for the instruction cache unit (ICU). The instruction-side PLB interface should only perform reads with eight-word cache line sizes.

During line reads, the address put out by the ICU is the required target word. Because the ICU interface can only provide word addresses, the lower two bits are tied Low at the chip level. For optimal performance, the target slave can provide either the target word first (aligned to a quadword containing the target word) or the data sequentially. In any case, the slave must use the Sl_rdWdAddr signal to indicate the word being transferred, which is used by the processor to correctly align data in its cache-fill buffer.

The ICU interface supports up to three outstanding transactions.

Data-Side Write (DCUWR)

The processor uses the Data-Side Write PLB (DCUWR) interface to perform write transfers with memory for the Data Cache Unit (DCU).

There are three types of transfers that occur via the DCUWR interface:

- Word transfers
- Four-word line transfers
- Eight-word line transfers

The address put on the PLB interface is the target byte address required by the DCU. For line transfers, the processor puts its data in sequential address order rather than target word first as done in data cache reads.

APU non-cacheable requests are performed with four-word line requests while all cacheable requests occur as eight-word line requests. The address output by the DCUWR interface is the target byte address of the data to be written.

Data-Side Read (DCURD)

The processor uses the Data-Side Read PLB (DCURD) interface to perform read transfers by the data cache unit.

Three types of transfers occur via the DCURD:

- Word transfers
- Four-word line transfers
- Eight-word line transfers

The address put on the PLB interface is the target byte address required by the DCU.

APU non-cacheable requests are performed with four-word line requests while all cacheable requests occur as eight-word line requests. The address put by the DCURD interface is the target byte address of the data to be read. For optimal performance in line reads, the slave must put out the target word first; however, in any case, it should indicate which word it is providing on the Sl_rdWdAddr signal.

FPGA Logic to Crossbar SPLB Interfaces

The identical SPLB 0 and SPLB 1 interfaces allow masters on the FPGA logic to access the slave buses on the crossbar via an FPGA logic arbiter. Although the FPGA logic masters are allowed any size of transactions included in the PLB specification, the SPLB interfaces convert some of these transactions to a subset of the PLB specification to optimize the crossbar transfer rate. All transactions from the SPLB to the crossbar are 128 data bits wide.

The types of transfers to the crossbar that can originate from the SPLBs are:

- Single transfers
- Quadword line transfers
- Eight-word line transfers
- Quadword fixed length bursts from a minimum length of 2 to a maximum length of 16.
- Variable length bursts or early terminated bursts from masters connected to their PLB.

  These variable length transactions are converted into a series of fixed length transactions. The size of the fixed length transaction is configurable through the PLB slave [0:1] configuration registers, CFG_PLBS0 and CFG_PLBS1.

Slave Port PLB Busy Signals

Generic PLB System

PLB masters receive a busy signal from the arbiter, which signifies that at least one of their acknowledged transactions has not been completed by the slave. The slave asserts an individual busy signal for each master attached to the arbiter.

- For a read, the slave asserts the busy signal one clock cycle after a command has been acknowledged until the last rddack for that transfer.
- For a write, the slave asserts the busy signal one clock cycle after a command has been acknowledged and past the last WrDack for that transfer. Write transaction completion by the slave might not coincide with the completion of the data being transferred to the slave. The slave can take additional cycles to complete the transaction after all the data has been transferred from the master. This is common in

situations where the slave has an internal queue to perform transfers to the physical storage. The busy signal thus notifies the master whether the data has been transferred to its final destination by the slave, which can prevent data coherency problems in systems.

Embedded Processor Block

The embedded processor block in Virtex-5 FPGAs changes the definition of the Mbusy PLB signal because of system architecture constraints. The Mbusy signal to an FPGA logic or PLB master is asserted when its request is acknowledged. Deassertion of the busy signal depends on the destination of the transaction:

- Destination MPLB

  The busy signal is deasserted when the transaction has left the MPLB for the FPGA logic. Thus the busy signal is deasserted when the data phase of the transaction has completed to the FPGA logic slave. There is a delay from the last WrDack on the MPLB to FPGA logic bus to the deassertion of the busy signal to the master of several interconnect clock cycles. If the FPGA logic slave continues to assert its busy signal past the end of the data phase of a write, that signal is not propagated back to the originating master. If the busy signal to the master is required to mirror the busy signal from the slave, then TAttribute [7] should be set along with the request. See "Sync TAttribute," page 50 for more details.

- Destination MCI

  The busy signal is deasserted when the transaction leaves the crossbar for the MCI. Both the crossbar and the MCI are internal to the processor block. Because the MCI does not contain a busy signal, the concept of a FPGA logic slave being busy, as in the MPLB case, does not exist for the MCI.

The FPGA logic PLB slaves are attached to the MPLB via a soft arbiter, and the MPLB is a single PLB master to that arbiter. PLB arbiters assert a single busy signal to each master to inform the master that at least one of its transactions is outstanding in a slave. The arbiter thus sends one PLB_MBusy signal to the MPLB to notify it if any slaves are asserting their busy signal for the MPLB. The MPLB transaction might have come from one of several masters connected to the SPLB, the DMA, or the processor PLB interfaces. The MPLB cannot demultiplex the busy signal from the arbiter to the individual masters because the identity of the master that originated the transaction was lost when the data phase completed.

Figure 3-1 shows the busy signals in the FPGA logic and processor block. The busy signal from the FPGA logic is of little use to the embedded processor block and ignores it under normal transaction request. The PLB_MBusy signal sourced from the processor block to FPGA logic masters is based on the transaction leaving the processor block (Destination MPLB) or the transaction leaving the crossbar for the MCI (Destination MCI).

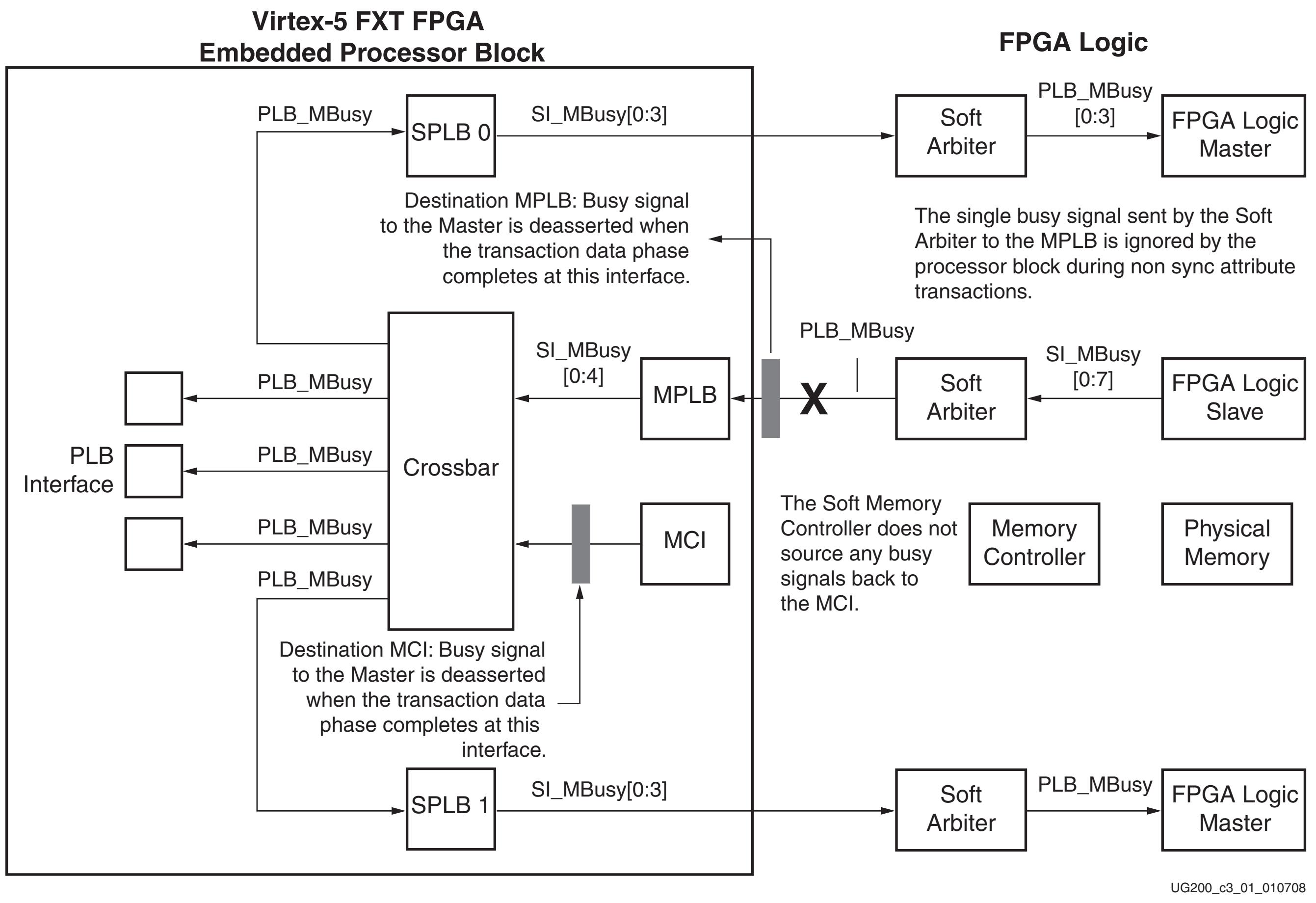

Virtex-5 FXT FPGA
Embedded Processor Block
FPGA Logic
PLB_MBusy
SPLB 0
SI_MBusy[0:3]
Soft Arbiter
PLB_MBusy [0:3]
FPGA Logic Master
Destination MPLB: Busy signal to the Master is deasserted when the transaction data phase completes at this interface.
The single busy signal sent by the Soft Arbiter to the MPLB is ignored by the processor block during non sync attribute transactions.
PLB_MBusy
SI_MBusy [0:4]
MPLB
X
Soft Arbiter
SI_MBusy [0:7]
FPGA Logic Slave
PLB Interface
PLB_MBusy
PLB_MBusy
Crossbar
PLB_MBusy
MCI
The Soft Memory Controller does not source any busy signals back to the MCI.
Memory Controller
Physical Memory
PLB_MBusy
Destination MCI: Busy signal to the Master is deasserted when the transaction data phase completes at this interface.
SPLB 1
SI_MBusy[0:3]
Soft Arbiter
PLB_MBusy
FPGA Logic Master
UG200_c3_01_010708

Figure 3-1: **Busy Signals in FPGA Logic and Embedded Processor Block**

Master Ports

MPLB

When the crossbar receives a request bound for the MPLB, it places the command in the command FIFO pending the start of the data phase.

Figure 3-2 shows the flow control exerted by the MPLB over the crossbar. If the command queue signal from the MPLB is asserted, the MPLB does not accept PLB address phases until at least one outstanding data phase is completed. The crossbar blocks its addrAck to the request that occurs when the MPLB command FIFO is full.

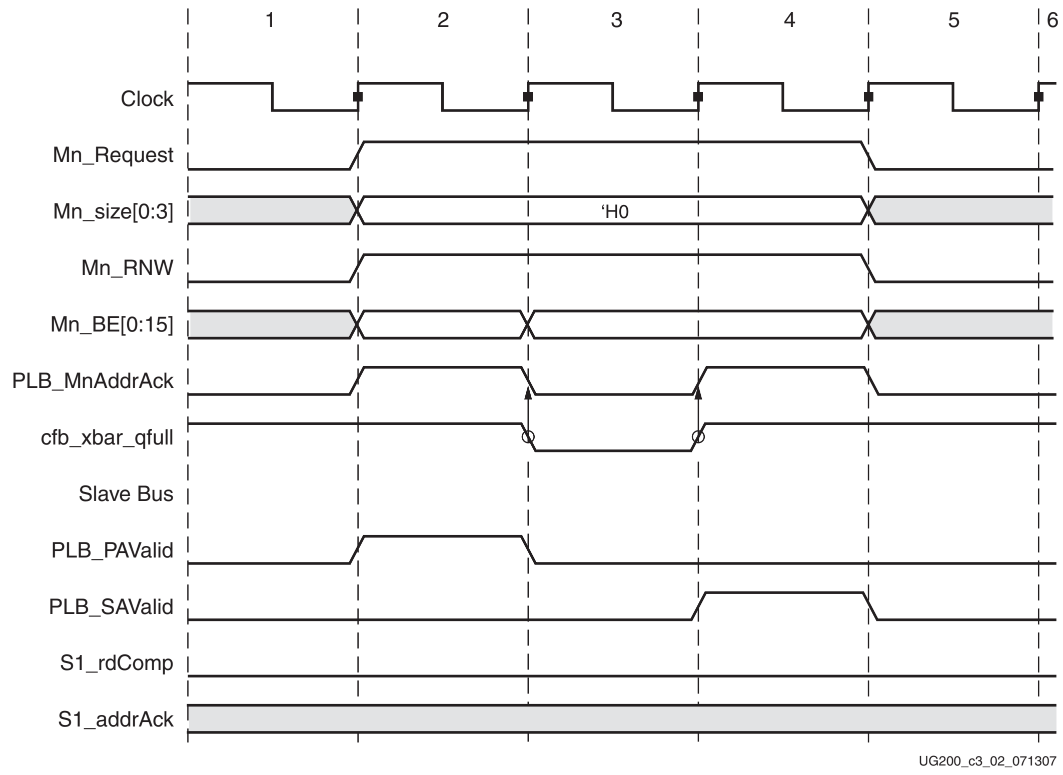


Figure 3-2: **MPLB Flow Control**

MCI

The memory interface is a fast, compact, and convenient way of connecting memory to the processor block. The memory interface is designed to be similar to a simple FIFO interface rather than the more complicated PLB interface. The interface consists of an address bus, two data buses (one for each direction), and a few control signals. All transactions to the FPGA logic have a constant length, greatly simplifying the design of the soft memory controller. Every transaction requires at minimum an address and a signal to indicate if it is a read or a write (MIMCREADNOTWRITE). If the transaction is a write, write data is presented on the write bus (MIMCWRITEDATA). If the transaction is a read, the MCI block expects the data along with a valid signal (MCMIREADDATAVALID) at some future point in time on the read bus (MCMIREADDATA). When a transaction begins, the MCI does not terminate it. The MCI block can turn off the byte enables so the writes become useless.

Although the physical data buses are 128 bits wide, the user can optionally downsize the bus. This option allows the portion of the memory controller in FPGA logic to not have to implement large muxes when the real memory is smaller then 128 bits. When the user selects a 32-bit bus and the MCI has 256 bits of data to transmit, the MCI sends eight 32-bit back-to-back words to the FPGA logic on bits 0 to 31 of the MIMCWRITEDATA bus. Not only does this save area in the FPGA logic, it also allows for higher speeds. This muxing is also implemented on the read path, so that the FPGA logic does not have to form 128-bit words for the MCI.

In addition to FPGA logic-side variable widths, FPGA logic burst lengths are also variable. Every time the burst length is reached, a new address is generated to send out to the FPGA logic. For instance, assume the starting address is 0, a burst of four 128-bit words (64 bytes) is to be sent to the FPGA logic portion of the memory controller, the MCI is set to 128 bits wide, and the burst length is 2. The transaction on the MCI is (address 0, write 0-15), (write 16-31), (address 32, write 32-47), and (write 48-63).

Variable burst lengths allow different memory controllers with different requirements to be attached to the MCI, while keeping the logic in the memory controller to a minimum. For example, one memory device might support bursts of 8, while another simpler memory device might support single word transactions (burst length equal to 1).

The MCI block takes the address directly from the crossbar and sends it to the FPGA logic, adjusting the address when required for bursts. The MCI block has no concept of what memory is actually connected up to it. Therefore, if a user writes to addresses `0x900` and the memory connected only has addresses from `0x000` to `0x7FF`, the memory at address `0x100` is overwritten. The MCI block assumes that the user of the system knows about this issue.

As noted in "FPGA Logic to Crossbar SPLB Interfaces," page 38, the MCI block does not accept indeterminate bursts. Instead, the MCI block relies on the PLB interface to break up the transactions into known fixed sized bursts. Because the processor does not create this type of transaction, this functionality was moved to the perimeter of the system, which also allowed higher bus utilization rates inside of the crossbar.

See Chapter 5, "Memory Controller Interface," for more details on this interface.

Disabling the MCI

The MCI can be disabled via a configuration bit. If the MCI block is disabled, any transaction directed to the MCI from the processor PLB interfaces causes the time-out signals from the crossbar to the PLB interface to be asserted. The SPLB blocks access to the MCI if it is disabled, and hence SPLB transactions need not be blocked.

Interrupts

The crossbar is the central location where the distributed PLB interrupts are collected. The crossbar sees the input interrupts as either level signals (which it does not latch) or edges (which it has to latch internally). The level interrupt signals are latched in the originating block. The crossbar contains the DCR interface logic to read the interrupts and clear them. The crossbar also asserts a global interrupt signal, PPCEICINTERCONNECTIRQ, when any non-masked interrupt is asserted.

The interrupts are cleared by writing a one to the interrupt bit of the register via DCR access. If the crossbar internally latches that interrupt, the write to the bit clears the interrupt bit. If the interrupt is latched in another block, the crossbar sends an interrupt clear signal to that block to instruct the block to reset it.

Functional Description

Arbitration

The crossbar separately arbitrates for the two slave buses (via the MCI and MPLB ports) between the five master PLB buses (via the SPLB 0, SPLB 1, ICURD, DCUWR, and DCURD ports) and grants slave bus access to the winning master. Transactions can be initiated on the two slave buses simultaneously. Which bus wins arbitration depends on:

- Priority
  - Request priority
  - Master priority
- Lock status of the slave bus
- Transaction ordering requirement
- Abort status of master requests
- State of the crossbar command queues
- Pipelining turned off or on
- MCI enabled or disabled

Priorities

If multiple master requests access the same slave bus simultaneously, the crossbar uses two levels of priority arbitration to determine the order of master accesses on the slave bus.

- Request Priority (Level 1 arbitration)

  The priority bits (Mn_priority [0:1]) sent out by the masters on the PLB along with the request are the first priorities taken into consideration. If access order can be resolved using those priority bits, then no further arbitration takes place. When the PLB transactions are destined for the MCI, this first level of arbitration can be enabled or disabled by a bit value in bit 24 of DCR MI_CONTROL. See Table 5-1, page 135 for more information.

- Master Priority (Level 2 arbitration)

  If the request priorities of two or more masters are the same, the crossbar implements a second level of priority checking using master priorities, which is set up by the FPGA designer via the DCRs. The master priorities are distinct values. Thus the priorities are never the same between two masters.

  Refer to "0x23: Crossbar for PLB Master Arbitration Configuration Register (ARB_XBC), R/W," page 61 for further details.

Request Priority Level

The five PLB master buses requesting accesses through the crossbar specify the priority of each request through the request priority bus, Mn_priority[0:1]. In this two-bit bus, `2'b11` is the highest priority and `2'b00` is the lowest priority in a fixed priority system. For the SPLBs, the request priority bus value is set by the master that initiated the SPLB request to the crossbar. For the three processor PLB interfaces, the priority bits are determined by the value of the priority attributes on the embedded processor.

On simultaneous accesses to the same slave bus, the crossbar grants access to the master with the highest request priority.

Master Priority Level

If two or more requesting masters have the same request priority, the crossbar uses a master priority level to break the tie. Users set an arbitration scheme and each master's initial priority via a Device Control Register (DCR). The crossbar contains one DCR per slave bus, which stores the master priority information in the crossbar. Each master can be assigned a priority using a three-bit priority field. Valid values are from `3'b000` (lowest priority) to `3'b100` (highest priority) for the five masters. *Each master should have a distinct priority from the permissible values*. Unpredictable results occur when using priorities from `3'b101` to `3'b111` or priorities not distinctly different.

Users choose one of three arbitration schemes:

- Least Recently Used (LRU)
- Round Robin (RR)
- Fixed

The crossbar also maintains an internal copy of the priority values that can change from the initial set value after each access. For all arbitration schemes, the crossbar allows the highest priority master to access the slave bus. The priority value used for the arbitration is the internal value.

Each slave bus has an arbitration register, the MPLB (ARB_XBC) and the MCI (ARB_XBM), containing the priorities for each master. These registers get their default values from attributes on the processor block. These registers can be read and written to via DCR operations.

Least Recently Used Arbitration

In the LRU arbitration scheme (see Figure 3-3), the master with the highest internal priority is granted access to the slave bus. Internal master priorities are reassessed after each slave bus access even if arbitration was based on request priority. The crossbar maintains a queue with initial entries set up by the DCR arbitration register. When a master is granted access to the slave bus, it is removed and placed at the back of the queue. All masters lower in internal priority to the one granted access are moved up by one position. This priority scheme gives fair access for masters to the slave bus.

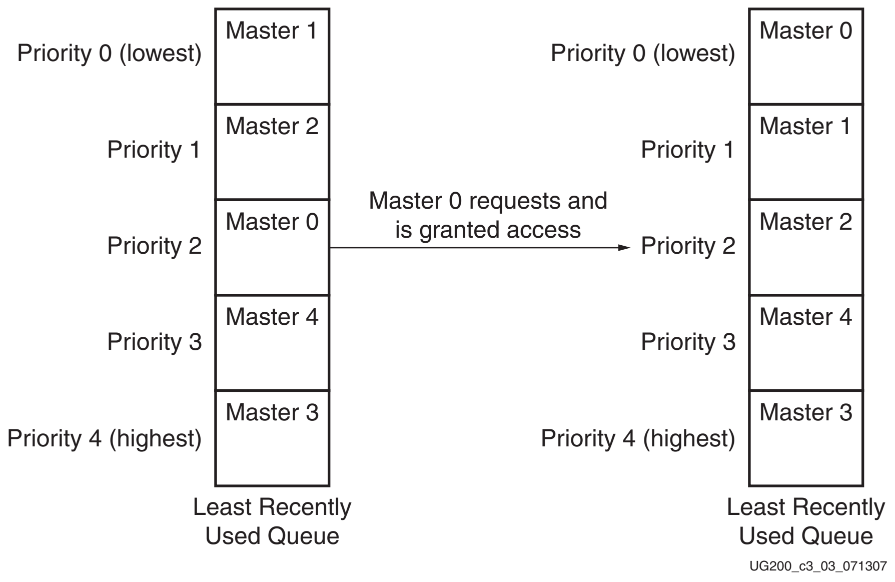


Figure 3-3: **LRU Arbitration**

Round-Robin Arbitration

In the round-robin arbitration scheme (see Figure 3-4), the master with the highest internal priority is granted access to the slave bus. Internal master priorities are reassessed after each slave bus access. The crossbar maintains a circular structure with five nodes, one for each master attached to the crossbar. The masters are initially assigned priorities based on the DCR arbitration register set up by the user. When a master gets access to the slave bus, the master priorities are rotated such that the master granted access is now the lowest priority master (`4'h0`). This priority scheme gives fair access for masters to the slave bus. In the unlikely event that the MPLB interface's read address pipelining feature (DCR `0x44` bit 26) must be disabled, which can cause a severe degradation in performance, avoid using the round-robin arbitration scheme because a traffic starvation issue can result.

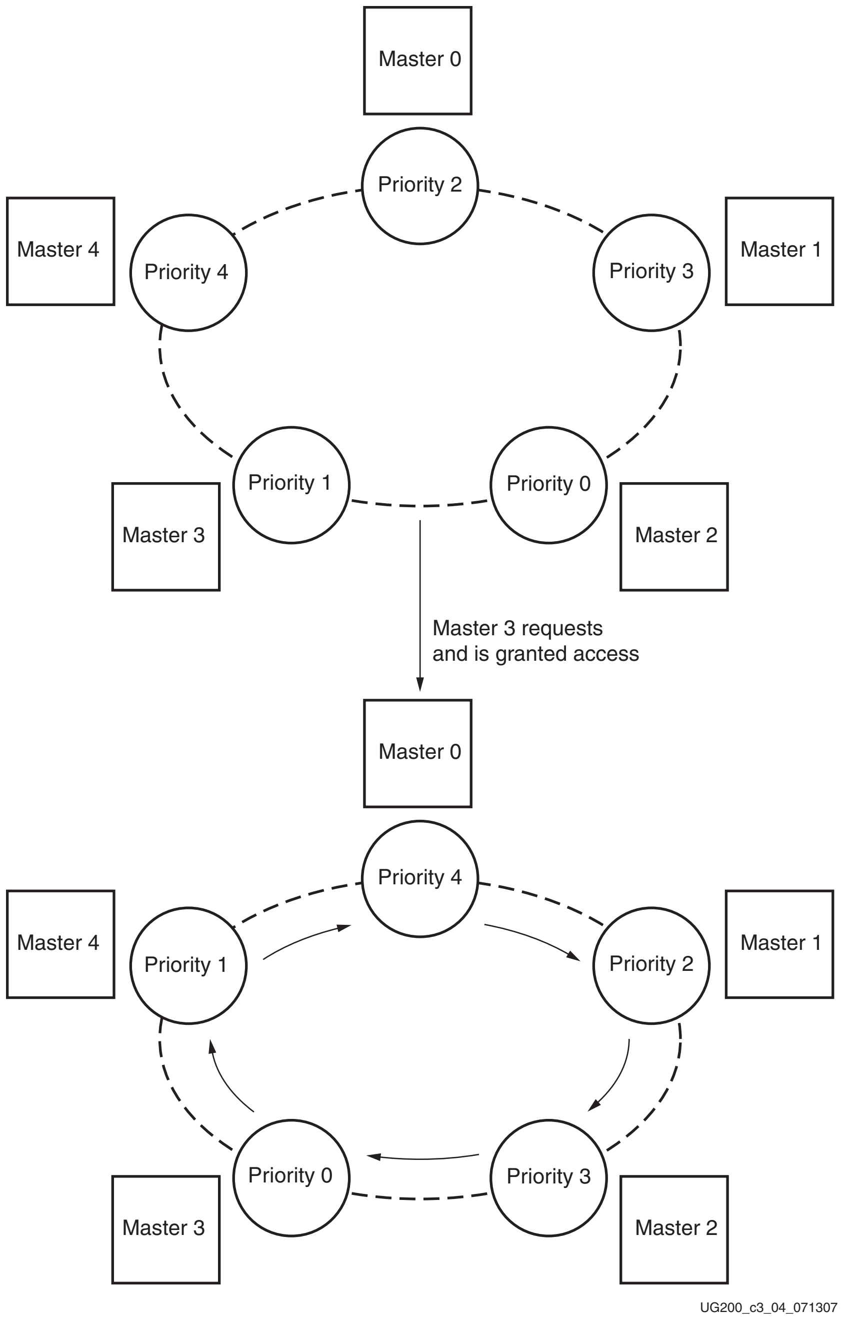


Figure 3-4: **Round-Robin Arbitration**

Fixed Priority

In the fixed priority scheme, the initial priorities set by the user remain unchanged during crossbar operation. The crossbar allows the highest priority master to access the slave bus. The arbitration scheme can cause starvation of lower priority masters, if higher priority masters continually request access to the same slave bus for prolonged periods of time.

DCR Arbitration Registers

The DCR arbitration registers consist of five three-bit priority fields (one for each master attached to the crossbar) and a two-bit arbitration mode field. The priority fields set up the initial master priority of the masters. Although the internal priorities can change during slave accesses depending on the arbitration mode (selected by the mode field), the priority values set in the DCRs do not change during crossbar operation. When a DCR arbitration register is read, it returns the original written or tied values, not the internal priorities used for arbitration.

Locked Transfers

Masters typically use locked transfers to guarantee an atomic sequence of transactions on the bus. The crossbar guarantees that the atomic sequence of operations requested by a master connected to one of the SPLB ports is preserved on the destination bus (MPLB).

The master asserts Mn_busLock with the request signal, which is sampled by the crossbar in the same cycle that the Sl_addrAck signal is asserted on the slave bus connected to the MPLB port. If both read and write buses are idle, the crossbar locks the slave bus for the requesting master. If there are outstanding transactions on the slave bus, the crossbar holds off assertion of the PLB_PAValid signal for the locked transfer until the slave bus is free. The lock extends to both the read and write sections of the slave bus. The locking master maintains its lock by asserting Mn_busLock until required. The request signal does need not be continually asserted during the lock period. It is only asserted when a transfer is required.

To prevent deadlock scenarios under certain traffic conditions, after the crossbar has granted a lock request, it blocks further requests from other PLB masters—regardless of where the targeted slave is.

The crossbar gets the Mn_busLock signal from the SPLB 0 or the SPLB 1. The processor PLB interfaces do not implement the Mn_busLock signal. If the locking master is the highest priority requesting master, the crossbar grants exclusive access to the locking master until the Mn_busLock signal is released. If requests are pending on the slave bus when a winning lock request is made, the crossbar ignores all other master requests until the lock request is granted by the crossbar and released by the master. This behavior prevents deadlock conditions where higher priority masters get access to the slave bus while the crossbar holds off the PLB_PAValid of the locking master request during completion of pending transactions.

If the locking master releases Mn_busLock for one clock cycle, the crossbar no longer locks the slave bus for that master but arbitrates in the normal manner on the next clock cycle. When the Mn_busLock signal is released, the crossbar continues to block all master requests until the pending transfers to that master are completed.

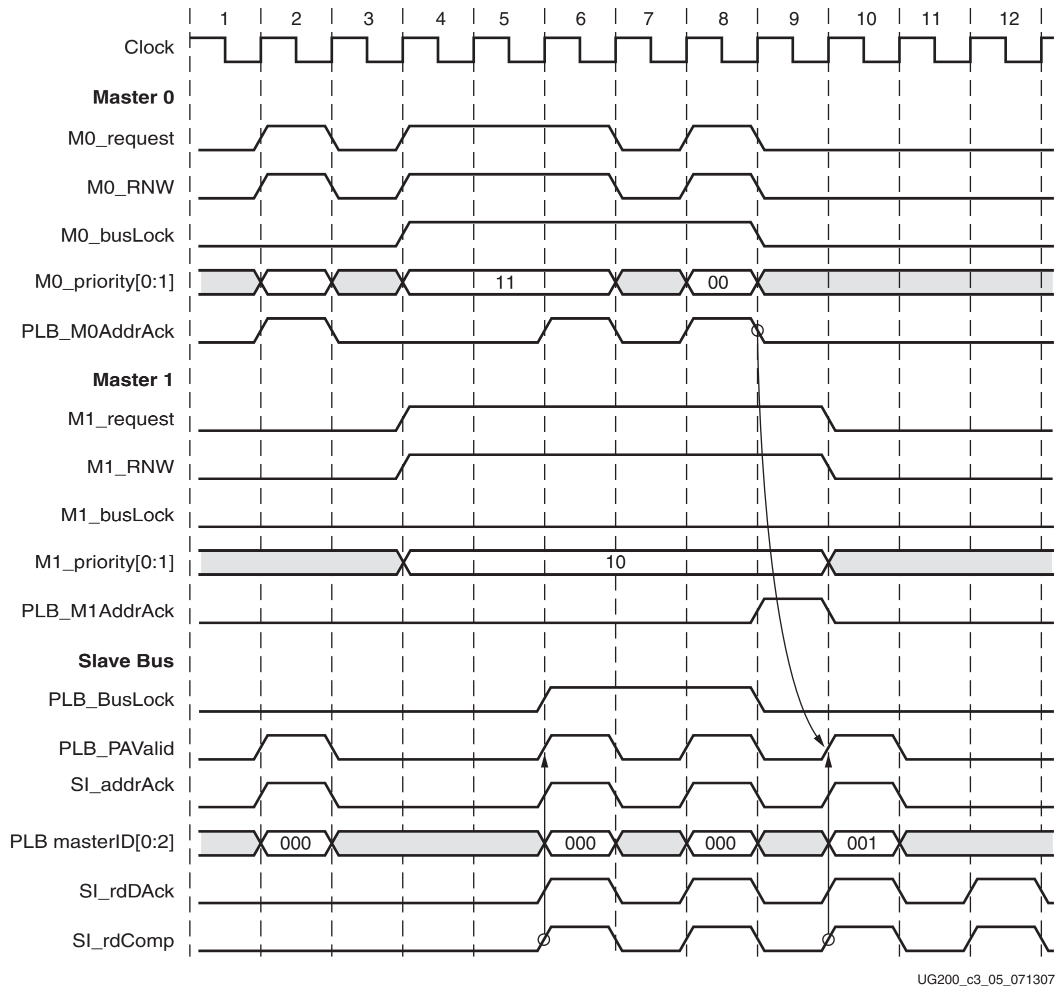


Figure 3-5: **Locked Transfer Waveforms**

Figure 3-5 shows the waveforms for a locked transfer operation. The cycles are defined below:

- In cycle 0, both the master and slave buses are in idle state.
- Master 0 requests a non-locked single read transfer in clock cycle 2 and is granted access to the slave bus.
- In clock cycle 4, Master 0 requests a *locked* single read transfer, and Master 1 requests a non-locked single read transfer. Because the request priority of Master 0 is higher than Master 1, Master 0 wins the arbitration. However, the pending transfer from cycle 2 blocks the assertion of PLB_PAValid for the locked Master 0 request until the transfer completes in cycle 6.
- When Sl_rdComp is asserted in cycle 6, the crossbar asserts PLB_PAValid and PLB_BusLock. The slave bus is now locked for exclusive access by Master 0, and the Master 1 request is ignored.
- In cycle 8, Master 0 requests with a priority of 0 (lower priority than Master 1) and gets access to the slave bus because it has exclusive access to the bus.
- When M0_busLock is released in cycle 9, the crossbar waits for all pending transactions on the slave bus to complete, which takes place in cycle 10. The crossbar then arbitrates in the normal manner and grants slave bus access to Master 1.

Ordering Requirement of Transactions in the Crossbar

From the master's perspective, the PLB protocol expects the data phase of transfers in the same direction (read or write) to occur in the order they were originally issued. This operation is essential because the individual data phases do not contain any transfer identification, and the requesting master assumes that the occurring data phase is for the oldest pending transfer. Because the data phase for transfers in different directions occurs on separate buses, ordering between reads and writes is not required. In a single slave bus PLB system, the arbiter need not be concerned with this ordering requirement, because the arbiter simply forwards the transfer request to the slave responsible for ordering the data phases.

The more complex crossbar has two slave buses, which make the system susceptible to order mismatch even if the PLB slaves conform to the ordering requirement. If a master issues two consecutive requests (one to the MPLB and one to the MCI) in the same direction (either both reads or both writes), the crossbar must hold off sending the second request to the slave until the first transaction completes. A situation might arise where the slave command queue in the crossbar is full for the first transfer and empty for the other transfer. In this situation, if ordering is not maintained by the crossbar, the slave of the second request might start its data phase earlier than the slave of the first request. This master wrongly assumes the incoming data is for the first request, creating a data mismatch for the requesting master.

The crossbar implements data phase ordering as follows:

- If a master request is granted to either slave bus, a request of the same direction to the other slave bus is blocked from arbitration consideration until the previous request is completed.
- Requests to the same slave bus are not blocked if there are pending requests in the same direction. In this case, ordering is maintained in the command FIFO and then by the slave.
- Transactions from the same master in the opposite direction of the pending transfer are not blocked from being issued to the other slave bus.
- While a master is blocked from arbitration consideration due to the ordering requirement, the crossbar arbitrates normally without the blocked master and grants access to the highest priority master.

The two arbiters within the crossbar use seven signals to communicate with each other:

- Busy signal for the ICURD interface of the embedded processor
- Busy signal for the DCUWR interface of the embedded processor
- Busy signal for the DCURD interface of the embedded processor
- Read busy signal for the SPLB 0 interface
- Write busy signal for the SPLB 0 interface
- Read busy signal for the SPLB 1 interface
- Write busy signal for the SPLB 1 interface

These busy signals notify the other arbiter about the master and direction of any pending transfers. One arbiter can then block arbitration of transfers from any master with pending transfers in the same direction on the other arbiter.

Sync TAttribute

A master might require notification when a particular write transaction is complete not just on the bus but also on the output of the module that is connected to the bus (for example, on the physical memory connected to a memory controller module). In generic PLB, the slave notifies the master of this by asserting its busy signal for that master. The slave can continue to assert its busy signal for a transaction past when its data phase has completed on the PLB. Figure 3-6 shows this situation where the busy signal sourced from the MPLB to the crossbar is based on transaction completion at the FPGA logic interface. In the embedded processor block, this generic PLB requirement is not met because with the completion of the data phase, the crossbar might end the connection and connect up a new master and slave. The Sync TAttribute feature facilitates the generic notification requirement without using the busy signal mechanism.

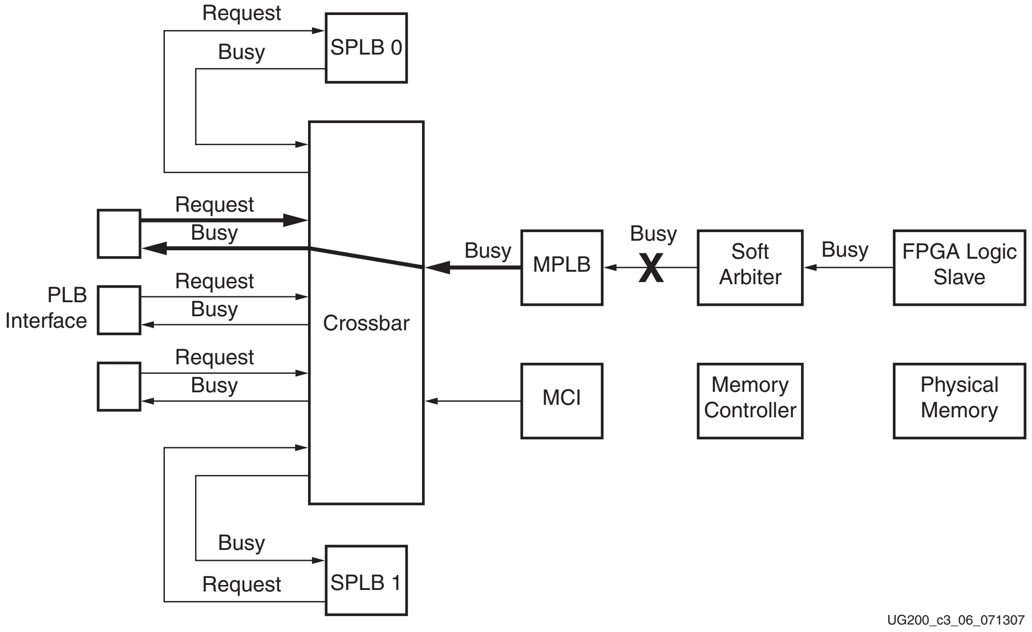


Figure 3-6: **Nonsync TAttribute Situation**

The Sync TAttribute feature is important in at least one scenario, where there is a device on the MPLB whose interrupt register bit is to be cleared. The embedded processor writes to the device's interrupt register to clear the interrupt. When the busy signal for that transaction goes Low, the embedded processor re-enables the interrupt. Without the TAttribute function, the clearing of the interrupt could be posted within the embedded processor block and might not have completed through to the target slave. In this scenario, if the processor re-enables the interrupt before the posted write has gone through, the embedded processor is wrongly interrupted for the second time.

If a master requires that the busy signal sourced to it mirrors (with a delay in deassertion) that busy signal coming from its target slave, the master should request the TAttribute [7] bit be set in the transaction, making this a Sync TAttribute transaction.

On this sync transaction request from a master winning arbitration:

- The crossbar blocks subsequent requests from all five master buses until the sync transaction and all transactions preceding it in the MPLB FIFO complete. This condition ensures that the busy signal from the FPGA logic slave to the master is deasserted for the sync transaction completion only because no other transaction can be pipelined behind it.

- The blocking takes place only after the master requesting the sync transaction has won arbitration to the MPLB slave bus.
- The busy signal to the master requesting a Sync TAttribute transaction is asserted until the slave destination deasserts its busy signal. Thus the FPGA logic busy signal is propagated through to the crossbar during a sync transaction.

Figure 3-7 shows the situation implementing a Sync TAttribute bit. When the Sync TAttribute (bit 7) of a request winning arbitration is set, the crossbar blocks all subsequent requests. The MCI and MPLB now propagate the FPGA logic busy signal all the way through the crossbar to the respective masters.

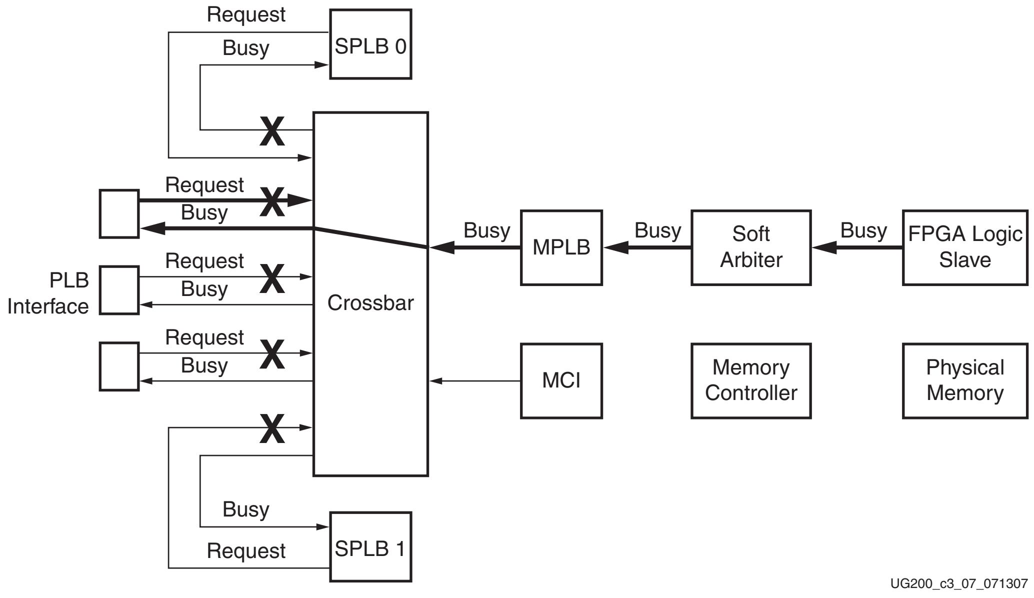


Figure 3-7: **Sync TAttribute Situation**

The master requesting the sync transaction can either be on the SPLB or can be one of the processor PLB interfaces. The processor can set the Sync TAttribute bit by setting it in the translation lookaside buffer (TLB) entry for the page of memory where the data is located. All transactions to that page (data loads/stores) are issued by the processor PLB master interface with the Sync TAttribute bit asserted. The Sync TAttribute bit for that page is set in software using user-defined storage attribute bit 3 (U3) via the tlbwe instruction. For FPGA logic PLB masters sitting on the SPLB interface, the master can set the attribute while making a request to the SPLB through the soft arbiter. The SPLB buffers the command in an internal FIFO and propagates the Sync TAttribute bits to the crossbar.

Address Mapping

The crossbar accepts requests from five PLB master buses and funnels them to either the MPLB slave bus or the MCI slave bus based on an address map programmed by the user. Masters on the PLB have access to a 36-bit address bus consisting of a 4-bit upper address bus, UABus, and a 32-bit lower address bus. (The 4-bit upper address bus is defined to be zero within the EDK tools and IP.) The 36 address bits cover a 64 GB address range. The 32-bit lower address bus covers a 4 GB address range.

The address map is programmed by:

1. A 32-bit template selection register (TMPL_SEL_REG)
2. Four 32-bit address template registers (TMPLx_XBAR_MAP)

These registers, which are DCR programmable, get their default values from attribute pins on the processor block.

The TMPL_SEL_REG register is a 32-bit register that concatenates 16 2-bit values. When it receives a request, the crossbar examines the four upper address bits, UABus [28:31], and indexes into the TMPL_SEL_REG register to obtain a two-bit value. The two bits determine which of the four address template registers to used for the request (`2'b00` = TMPL0_XBAR_MAP, ..., `2'b11` = TMPL3_XBAR_MAP).

EDK sets TMPL_SEL_REG so that only the TMPL0_* registers are used to define crossbar address mapping and SPLB address decoding for the supported 4 GB address space. This is the only supported register configuration.

The four address template registers (TMPLx_XBAR_MAP) are 32 bits wide, where each bit represents 1/32 of a 4 GB address space (128 MB). The crossbar uses the most-significant five bits of the lower address bus (Abus[0:4]) to index into the selected address template register to obtain a single bit value. If the bit in the address template register is set, a request to that 128 MB address space is directed to the MCI slave bus. If it is cleared, the request is directed to the MPLB slave bus.

Example Configuration

Assume the four address template registers have the following values:

TMPL_SEL_REG = `32'h3FFF_FFFF`

TMPL0_XBAR_MAP = `32'h0000_000F`

TMPL1_XBAR_MAP = `32'h0000_0000`

TMPL2_XBAR_MAP = `32'h0000_0000`

TMPL3_XBAR_MAP = `32'h0000_0000`

1. If the request address is `36'h0_E923_3245`:

   First an address template register needs to be selected. Because the four upper address bits are `4'h0`, two bits are used starting at bit position 0. In the current configuration, the bits are `2'b00`. Therefore TMPL0_XBAR_MAP is used to decode the remaining address bits.

   The five most-significant address bits are used to index into the TMPL0_XBAR_MAP register. Here the five bits are `5'b11101` or `5'd29`. TMPL0_XBAR_MAP[29] is `1'b1` (set). Therefore this address is destined for the MCI.

2. If the request address = `36'h0_ABCD_9872`:

   Again, TMPL0_XBAR_MAP is used to decode the lower address bus bits.

   The five most-significant address bits are used to index into the TMPL0_XBAR_MAP register. Here the next lower five bits are `5'b10101` or `5'd21`. TMPL0_XBAR_MAP[21] is `1'b0` (cleared). Therefore this address is destined for the MPLB.

Figure 3-8 shows the Address Template register, which divides the 4 GB address space into 128 MB regions. If the bit corresponding to a 128 MB region is set, that request is forwarded to the MCI; otherwise, it is sent to the MPLB.

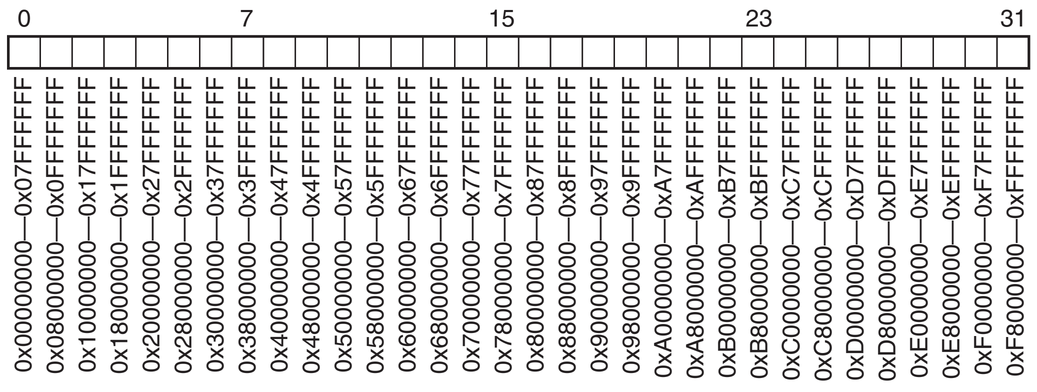


Figure 3-8: **Address Template Register**

The crossbar implements the address mapping for the processor PLB interfaces. For requests originating from the SPLBs, the SPLBs predecode the requests. The SPLBs provide this information to the crossbar via the MPLBnMCI signal. Thus the crossbar does not have to decode the address again, which improves timing because the address-mapping logic falls on the critical path.

Pipelining

Command Queues

The PLB protocol defines two phases of a transaction, the address phase and data phase, which occur independently on separate buses. When a request is acknowledged by a slave, the slave commits to provide the data during the data phase of the transaction. A delay might occur before the data phase is started by the slave. Multiple data phases can also be pipelined by the slaves in the system. To keep track of the outstanding data phases, the crossbar maintains two command queues for each arbiter, one for reads and the other for writes. The crossbar uses the information stored in the command queues to direct the slave data phase responses to the appropriate master and to determine if a particular transaction has completed.

The read and write command queues are each five deep and hence up to five read and five write data phases can be outstanding to any slave bus.

When a master requests a transaction, the crossbar arbitrates between it and other requesting masters. A winner is ascertained within the same clock cycle, the request is put on the slave address bus, and the byte enable (BE) and size transfer qualifiers are pushed into the crossbar command FIFO.

When a data phase completes, the command FIFO is popped and used to redirect the slave signals to the correct masters.

Miscellaneous Notes

Conversion of Quadword Line Transfers to Single Transfers

Any quadword transfer requested from the crossbar is converted to a single transfer at the slave port. Because the internal crossbar switch data buses are always 128 bits wide, single and quadword line transfers are equivalent.

Miscellaneous Signals

PPCCPMINTERCONNECTBUSY

The PPCCPMINTERCONNECTBUSY output from the processor block is asserted when there are any *transactions in flight* within the processor block interconnect. The MPLB, MCI, DMA, SPLB, and DCR interface blocks generate independent busy signals that are ORed together to create PPCCPMINTERCONNECTBUSY.

Transactions in flight are defined as:

- SPLB
  - Acknowledged request from the FPGA logic master to the SPLB that has not completed its data phase
  - Posted writes present within the SPLB
  - A pending transaction between the SPLB and crossbar
- MPLB
  - Acknowledged request from the MPLB to a FPGA logic slave whose data phase has not been completed
  - Posted writes present within the MPLB
  - A pending transaction between the crossbar and MPLB
- MCI
  - Posted writes present within the MCI
  - A pending transaction between the crossbar and MCI
- DMA
  - A pending transaction on the RX local link
  - A pending transaction on the TX local link
  - A pending transaction between the DMA and the SPLB/crossbar
  - The RX descriptor chain is not complete
  - The TX descriptor chain is not complete
- DCR
  - A pending transaction on the DCR bus

Two scenarios where the PPCCPMINTERCONNECTBUSY signal can be used are:

- Sleep Control Logic

  In a typical system, the clock control module puts the system into sleep mode by gating the clock, when all the system's masters have asserted their sleep request signals. However, there might still be posted writes present within the processor block that need to be flushed out to the slaves before the system is put to sleep. The clock control module thus looks at the PPCCPMINTERCONNECTBUSY signal in addition to the sleep request from the masters to gate the clock to the system. When the PPCCPMINTERCONNECTBUSY signal is deasserted, the clock control module knows it is safe to gate the clock without inducing any undefined behavior.

- Dynamic Reconfiguration of the Processor Block

  The configuration registers of the interconnect blocks can be written to by DCR transactions without resetting the processor block. This can only be done when there are no transactions in flight within the processor block, else the behavior is undefined.

The DCR master who performs the reconfiguration can sample the PPCCPMINTERCONNECTBUSY signal to determine if it is safe to reconfigure the processor block. If the PPCCPMINTERCONNECTBUSY signal is deasserted, the master can write to the configuration registers via DCR transactions to change parameters like crossbar priorities. This signal can also be used to safely change clock ratios interfacing to or within the processor block.

Usage Notes and Limitations

Crossbar Limitations for PCI and PCI Express Designs

A crossbar is defined to allow transactions between ports that cannot be monitored by devices on other ports. This can potentially lead to issues with ordering and forward progress that can in some cases cause livelock or deadlock. As described below, both PCI® and PCI Express designs require a specific system configuration to avoid these issues.

See the "PLB Interconnection Techniques," page 124 for more details.

Device Control Registers (DCRs)

Overview of the DCR Map

The crossbar includes a number of configuration bits that are accessible through the DCR interface. The embedded processor or the external DCR master can read or modify the crossbar configuration from the default value by issuing DCR read or DCR write commands.

After a DCR write transaction is requested, the crossbar performs the internal register write and asserts its acknowledge signal after a three-cycle delay. The crossbar similarly puts out the requested read data and asserts the acknowledge signal after a three-cycle delay. The acknowledge signal is deasserted three clock cycles after the read or write request signal is deasserted.

Detailed DCR Descriptions

DCRs for the PLB Interfaces and Crossbar (0x20 – 0x5F)

A block of 64 DCR locations (`0x20` to `0x5F`) is allocated for use by the crossbar, two PLB slaves (SPLB 0 and SPLB 1), the PLB master (MPLB), and the Address Map configuration registers. Four separate DCR lists are shown in Table 3-1 through Table 3-4.

All the interrupt status bits of the PLB interfaces and the crossbar are consolidated in the Interrupt Status register at `0x20`. Registers with a tie-off value can be set to specified default values via the corresponding attribute on the embedded processor block in Virtex-5 FPGAs.

Table 3-1: **List of DCRs for the Crossbar**

Address	Mnemonic	Description	Type
Global Configuration and Status			
`0x20`	IST	Interrupt Status Register	Clear on Write to bit and Read Only
`0x21`	IMASK	Interrupt Mask Register	R/W
`0x22`	-	Reserved	-
Crossbar for PLB Master Configuration			
`0x23`	ARB_XBC	Arbitration Configuration Register	R/W
Crossbar for PLB Master Status			
`0x24`	FIFOST_XBC	FIFO Overflow and Underflow Status	Clear on Write to bit
Crossbar for PLB Master Hardware Debug			
`0x25`	SM_ST_XBC	State Machine States Register	Read Only
`0x26`	MISC_XBC	Miscellaneous Control and Status	R/W, Write Only
`0x27`	-	Reserved	-
Crossbar for MCI Configuration			
`0x28`	ARB_XBM	Arbitration Configuration Register	R/W

Table 3-1: **List of DCRs for the Crossbar *(Cont'd)***

Address	Mnemonic	Description	Type
Crossbar for MCI Status			
`0x29`	FIFOST_XBM	FIFO Overflow and Underflow Status	Clear on Write to bit
Crossbar for MCI Hardware Debug			
`0x2A`	-	Reserved	-
`0x2B`	MISC_XBM	Miscellaneous Control and Status	R/W, Write Only
`0x2C`	-	Reserved	-
Address Map Configuration			
`0x2D`	TMPL0_XBAR_MAP	Template Register 0 for Crossbar	R/W
`0x2E`	TMPL1_XBAR_MAP	Template Register 1 for Crossbar	R/W
`0x2F`	TMPL2_XBAR_MAP	Template Register 2 for Crossbar	R/W
`0x30`	TMPL3_XBAR_MAP	Template Register 3 for Crossbar	R/W
`0x31`	TMPL_SEL_REG	Template Selection Register	R/W
`0x32`	-	Reserved	-
`0x33`	-	Reserved	-

Table 3-2: **List of DCRs for PLB Slave 0 (SPLB 0)**

Address	Mnemonic	Description	Type
Configuration			
`0x34`	CFG_PLBS0	Configuration Register	R/W
`0x35`	-	Reserved	-
Status			
`0x36`	SEAR_U_PLBS0	Slave Error Address Register, upper 4 bits	Clear on Write to `0x38`
`0x37`	SEAR_L_PLBS0	Slave Error Address Register, lower 32 bits	Clear on Write to `0x38`
`0x38`	SESR_PLBS0	Slave Error Status Register	Clear on Write
`0x39`	MISC_ST_PLBS0	Miscellaneous Status Register	Clear on Write to bit
`0x3A`	PLBERR_ST_PLBS0	PLB Error Status	Clear on Write to bit
Hardware Debug			
`0x3B`	SM_ST_PLBS0	State Machine States Register	Read Only
`0x3C`	MISC_PLBS0	Miscellaneous Control and Status	R/W, WO, RO
`0x3D`	CMD_SNIFF_PLBS0	Command Sniffer	R/W
`0x3E`	CMD_SNIFFA_PLBS0	Command Sniffer Address	R/W
`0x3F`	-	Reserved	-

Table 3-2: **List of DCRs for PLB Slave 0 (SPLB 0)** ***(Cont'd)***

Address	Mnemonic	Description	Type
Address Map			
0x40	TMPL0_PLBS0_MAP	Template Register 0	R/W
0x41	TMPL1_PLBS0_MAP	Template Register 1	R/W
0x42	TMPL2_PLBS0_MAP	Template Register 2	R/W
0x43	TMPL3_PLBS0_MAP	Template Register 3	R/W

Table 3-3: **List of DCRs for the PLB Slave 1 (SPLB 1)**

Address	Mnemonic	Description	Type
Configuration			
0x44	CFG_PLBS1	Configuration Register	R/W
0x45	-	Reserved	-
Status			
0x46	SEAR_U_PLBS1	Slave Error Address Register, upper 4 bits	Clear on Write to 0x48
0x47	SEAR_L_PLBS1	Slave Error Address Register, lower 32 bits	Clear on Write to 0x48
0x48	SESR_PLBS1	Slave Error Status Register	Clear on Write
0x49	MISC_ST_PLBS1	Miscellaneous Status Register	Clear on Write to bit
0x4A	PLBERR_ST_PLBS1	PLB Error Status	Clear on Write to bit
Hardware Debug			
0x4B	SM_ST_PLBS1	State Machine States Register	Read Only
0x4C	MISC_PLBS1	Miscellaneous Control and Status	R/W, WO, RO
0x4D	CMD_SNIFF_PLBS1	Command Sniffer	R/W
0x4E	CMD_SNIFFA_PLBS1	Command Sniffer Address	R/W
0x4F	-	Reserved	-
Address Map			
0x50	TMPL0_PLBS1_MAP	Template Register 0	R/W
0x51	TMPL1_PLBS1_MAP	Template Register 1	R/W
0x52	TMPL2_PLBS1_MAP	Template Register 2	R/W
0x53	TMPL3_PLBS1_MAP	Template Register 3	R/W

Table 3-4: **List of DCRs for the PLB Master (MPLB)**

Address	Mnemonic	Description	Type
Configuration			
`0x54`	CFG_PLBM	Configuration Register	R/W
`0x55`	-	Reserved	-
Status			
`0x56`	FSEAR_U_PLBM	FPGA Logic Slave Error Address Register, upper 4 bits	Clear on Write to `0x58`
`0x57`	FSEAR_L_PLBM	FPGA Logic Slave Error Address Register, lower 32 bits	Clear on Write to `0x58`
`0x58`	FSESR_PLBM	FPGA Logic Slave Error Status Register	Clear on Write
`0x59`	MISC_ST_PLBM	Miscellaneous Status	Clear on Write to bit
`0x5A`	PLBERR_ST_PLBM	PLB Error Status	Clear on Write to bit
Hardware Debug			
`0x5B`	SM_ST_PLBM	State Machine States Register	Read Only
`0x5C`	MISC_PLBM	Miscellaneous Control and Status	R/W, Write Only
`0x5D`	CMD_SNIFF_PLBM	Command Sniffer	R/W
`0x5E`	CMD_SNIFFA_PLBM	Command Sniffer Address	R/W
`0x5F`	-	Reserved	-

DCRs for the Crossbar (0x20 to 0x33)

0x20: Interrupt Status Register (IST), Clear on Writes, Read Only

This register contains all the interrupt status bits of the two PLB slave interfaces, PLB master interface, and the crossbar (see Table 3-5). All register bits are cleared on writes, except those that are marked as read only (RO). Writing a 1 to a clear-on-write bit clears it. The read-only bits are cleared by writing to their corresponding source DCRs. For example, bit 7 is cleared by writing 0s to the PLBS 0 FIFO Error Status register.

Note: Even if a particular interrupt is masked, these status bits are still set if the error condition is detected.

Table 3-5: **Bit Definitions for the IST Register**

Bits	Field	Type	Default	Description
0:2	Reserved	-	`000`	Reserved
3	INT_CFG_ERR_S0	RO	`0`	Configuration or command error, PLBS0. See register `0x39` for further information.
4	INT_MIRQ_S0	RO	`0`	PLB MIRQ error, PLBS0
5	INT_MRDERR_S0	Clr on Wr	`0`	Read transaction error, PLBS0. See registers `0x36` through `0x38` for further information.
6	INT_MWRERR_S0	Clr on Wr	`0`	Write transaction error, PLBS0. See registers `0x36` through `0x38` for further information.

Table 3-5: **Bit Definitions for the IST Register *(Cont'd)***

Bits	Field	Type	Default	Description
7	INT_FIFO_ERR_S0	RO	0	FIFO error interrupt, PLBS0. See register 0x39 for further information.
8:10	Reserved	-	000	Reserved
11	INT_CFG_ERR_S1	RO	0	Configuration or command error, PLBS1. See register 0x49 for further information.
12	INT_MIRQ_S1	RO	0	PLB MIRQ error, PLBS1
13	INT_MRDERR_S1	Clr on Wr	0	Read transaction error, PLBS1. See registers 0x46 through 0x48 for further information.
14	INT_MWRERR_S1	Clr on Wr	0	Write transaction error, PLBS1. See registers 0x46 through 0x48 for further information.
15	INT_FIFO_ERR_S1	RO	0	FIFO error interrupt, PLBS1. See register 0x49 for further information.
16	Reserved	-	0	Reserved
17	INT_CFG_ERR_M	RO	0	Configuration error, PLBM. See register 0x59 for further information.
18	INT_MIRQ_M	RO	0	PLB MIRQ error, PLBM
19	INT_MRDERR_M	Clr on Wr	0	Read transaction error, PLBM. See registers 0x56 through 0x58 for further information.
20	INT_MWRERR_M	Clr on Wr	0	Write transaction error, PLBM. See registers 0x56 through 0x58 for further information.
21	INT_ARB_TOUT_M	Clr on Wr	0	PLB Time-out error, PLBM
22	Reserved	-	0	Reserved
23	Reserved	-	0	Reserved
24	INT_FIFO_ERR_M	RO	0	FIFO error interrupt, PLBM. See register 0x59 for further information.
25	INT_FIFO_ERR_XM	RO	0	FIFO error, Crossbar for PLBM. See register 0x58 for further information.
26	INT_FIFO_ERR_MCI	RO	0	FIFO error, Crossbar for MCI. See register 0x5D for further information.
27:31	Reserved	-	0	Reserved

0x21: Interrupt Mask Register (IMASK), R/W

This register contains the interrupt mask information (see Table 3-6). Clearing a bit to 0 masks the interrupt generation from the corresponding interrupting source in register `0x20`.

Table 3-6: **Bit Definitions for the IMASK Register**

Bits	Field	Default	Description
0:2	Reserved	111	Reserved
3	M_INT_CFG_ERR_S0	1	Interrupt mask for configuration or command error, PLBS0
4	M_INT_MIRQ_S0	1	Interrupt mask for general error, PLBS0
5	M_INT_MRDERR_S0	1	Interrupt mask for read transaction error, PLBS0
6	M_INT_MWRERR_S0	1	Interrupt mask for write transaction error, PLBS0
7	M_INT_FIFO_ERR_S0	1	Interrupt mask for FIFO error, PLBS0
8:10	Reserved	111	Reserved
11	M_INT_CFG_ERR_S1	1	Interrupt mask for configuration or command error, PLBS1
12	M_INT_MIRQ_S1	1	Interrupt mask for general error, PLBS1
13	M_INT_MRDERR_S1	1	Interrupt mask for read transaction error, PLBS1
14	M_INT_MWRERR_S1	1	Interrupt mask for write transaction error, PLBS1
15	M_INT_FIFO_ERR_S1	1	Interrupt mask for FIFO error interrupt, PLBS1
16	Reserved	1	Reserved
17	M_INT_MPLB_ERR_M	1	Interrupt mask for configuration error, PLBM
18	M_INT_MIRQ_M	1	Interrupt mask for general error, PLBM
19	M_INT_MRDERR_M	1	Interrupt mask for read transaction error, PLBM
20	M_INT_MWRERR_M	1	Interrupt mask for write transaction error, PLBM
21	M_INT_ARB_TOUT_M	1	Interrupt mask for PLB time-out error, PLBM
22	Reserved	1	Reserved
23	Reserved	1	Reserved
24	M_INT_FIFO_ERR_M	1	Interrupt mask for FIFO error interrupt, PLBM
25	M_INT_FIFO_ERR_XM	1	Interrupt mask for FIFO error, Crossbar for PLBM
26	M_INT_FIFO_ERR_MCI	1	Interrupt mask for FIFO error, Crossbar for MCI
27:31	Reserved	5`b1	Reserved

0x23: Crossbar for PLB Master Arbitration Configuration Register (ARB_XBC), R/W

This register configures crossbar arbitration priority and mode operations (see Table 3-7). This register is initialized by embedded processor block attribute PPCM_ARBCONFIG. Arbitration priority values apply to fixed and round-robin arbitration only, with 4 corresponding to the highest priority and 0 to the lowest priority. Values between 5 and 7 are reserved and should not be used due to unpredictable behavior. The five device priority values must be mutually exclusive so that no two or more devices can have the same priority, otherwise there are unpredictable results.

Table 3-7: **Bit Definitions for the ARB_XBC Register**

Bits	Field	Default	Description
0:8	Reserved	`0`	Reserved
9:11	440ICUR	`100`	Instruction Read Priority
12	Reserved	`0`	Reserved
13:15	440DCUW	`011`	Data Write Priority
16	Reserved	`0`	Reserved
17:19	440DCUR	`010`	Data Read Priority
20	Reserved	`0`	Reserved
21:23	PLBS1	`000`	PLB Slave 1 Priority
24	Reserved	`0`	Reserved
25:27	PLBS0	`001`	PLB Slave 0 Priority
28	Reserved	`0`	Reserved
29	SYNCTATTR	`0`	Sync TAttribute (bit 7) enable, if set
30:31	MODE	`00`	Arbitration Mode. • `00`: For Least Recently Used (LRU) • `01`: For round-robin • `10`: For fixed priority • `11`: Reserved (should not be used, may lead to unpredictable behavior)

0x24: Crossbar for PLB Master FIFO Overflow and Underflow Status Register (FIFOST_XBC), Clear on Writes

This register indicates the FIFO overflow and underflow status for the PLB master (see Table 3-8). Individual register bits are cleared by writing 1s to them. Bit 31 of the interrupt register is set if any of the FIFO overflow or underflow bit is set. None of these bits should ever be set under normal operating conditions.

Table 3-8: **Bit Definitions for the FIFOST_XBC Register**

Bits	Field	Default	Description
0:27	Reserved	0	Reserved
28	FIFO_OF_RCMDQ	0	Indicates a write command queue overflow, when set
29	FIFO_UF_RCMDQ	0	Indicates a write command queue underflow, when set
30	FIFO_OF_WCMDQ	0	Indicates a read command queue overflow, when set
31	FIFO_UF_WCMDQ	0	Indicates a read command queue underflow, when set

0x26: Crossbar for PLB Master Miscellaneous Control and Status Register (MISC_PLBM), R/W or Write Only

This register contains miscellaneous control and status bits for the PLB Master (see Table 3-9). Read values for write-only bits are always 0s.

Table 3-9: **Bit Definitions for the MISC_PLBM Register**

Bits	Field	Default	Type	Description
0:29	Reserved	0	-	Reserved
30	FIFO_RCMDQ_RST	0	Write Only	Write a 1 to this bit to reset the Read Command Queue
31	FIFO_WCMDQ_RST	0	Write Only	Write a 1 to this bit to reset the Write Command Queue

0x28: Crossbar for MCI Arbitration Configuration Register (ARB_XBM), R/W

This register configures crossbar arbitration priority and mode operations (see Table 3-10). This register is initialized by embedded processor block attribute MI_ARBCONFIG. Arbitration priority values apply to fixed and round-robin arbitration only with 4 corresponding to the highest priority and 0 to the lowest priority. Values between 5 and 7 are reserved and should not be used due to unpredictable behavior. The five device priority values must be mutually exclusive so that no two or more devices can have the same priority, otherwise unpredictable results could occur.

Table 3-10: **Bit Definitions for the ARB_XBM Register**

Bits	Field	Default	Description
0:8	Reserved	0	Reserved
9:11	440ICUR	`100`	Instruction Read Priority
12	Reserved	0	Reserved
13:15	440DCUW	`011`	Data Write Priority
16	Reserved	0	Reserved
17:19	440DCUR	`010`	Data Read Priority
20	Reserved	0	Reserved
21:23	PLBS1	`000`	PLB Slave 1 Priority
24	Reserved	0	Reserved
25:27	PLBS0	`001`	PLB Slave 0 Priority
28:29	Reserved	0	Reserved
30:31	MODE	`00`	Arbitration Mode. • `00`: For Least Recently Used (LRU) • `01`: For round-robin • `10`: For fixed priority • `11`: Reserved (should not be used, may lead to unpredictable behavior)

0x29: Crossbar for MCI FIFO Overflow and Underflow Status Register (FIFOST_XBM), Clear on Writes

This register indicates the FIFO overflow and underflow status for the MCI (see Table 3-11). Individual register bits are cleared by writing 1s to them. Bit 31 of the interrupt register is set if the FIFO overflow or underflow bit is set. None of the bits should ever be set under normal operating conditions.

Table 3-11: **Bit Definitions for the FIFOST_XBM Register**

Bits	Field	Default	Description
0:27	Reserved	0	Reserved
28	FIFO_OF_RCMDQ	0	Write command queue overflow, when set
29	FIFO_UF_RCMDQ	0	Write command queue underflow, when set
30	FIFO_OF_WCMDQ	0	Read command queue overflow, when set
31	FIFO_UF_WCMDQ	0	Read command queue underflow, when set

0x2B: Crossbar for MCI Miscellaneous Control and Status Register (MISC_XBM), R/W or Write Only

This register contains miscellaneous control and status bits for the MCI (see Table 3-12). Read values for write-only bits are always 0s.

Table 3-12: **Bit Definitions for the MISC_XBM Register**

Bits	Field	Default	Type	Description
0:29	Reserved	0	-	Reserved
30	FIFO_RCMDQ_RST	0	Write Only	Write a 1 to this bit to reset the Read Command Queue
31	FIFO_WCMDQ_RST	0	Write Only	Write a 1 to this bit to reset the Write Command Queue

0x2D to 0x30: Crossbar Template Registers, R/W

There are four 32-bit template registers for the crossbar (see Table 3-13). Selection of one of the four registers for address mapping is done through the Template Selection Register. Each bit of a 32-bit register corresponds to 128 MByte address space for a total of 4 GB addressing. Traffic is routed to the MCI if the address is within the 128 MB address range that has the template bit set; otherwise the traffic is routed to the PLB Master. These registers are initialized by embedded processor block attributes XBAR_ADDRMAP_TMPL0 through XBAR_ADDRMAP_TMPL3.

Table 3-13: **Crossbar Template Registers**

Address	Mnemonic	Default	Description
`0x2D`	TMPL0_XBAR_MAP	`32'hFFFF_0000`	Template Register 0 for Crossbar
`0x2E`	TMPL1_XBAR_MAP	`32'h0000_0000`	Template Register 1 for Crossbar
`0x2F`	TMPL2_XBAR_MAP	`32'h0000_0000`	Template Register 2 for Crossbar
`0x30`	TMPL3_XBAR_MAP	`32'h0000_0000`	Template Register 3 for Crossbar

0x31: Template Selection Register (TMPL_SEL_REG), R/W

This register is the template selection register for specifying the address mapping template (see Table 3-14). There are 16 x 2-bit entries in this register corresponding to a 16 x 4-GB address space. Each two-bit field identifies which of the four TMPL*_XBAR_MAP registers are used to map crossbar addresses, which of the four TMPL*_PLBS0_MAP registers are used to enable address decoding on SPLB 0, and which of the four TMPL*_PLBS1_MAP registers are used to enable address decoding on SPLB 1. By default, all of these address template registers are configured so that template 0 controls all crossbar mapping and SPLB interface decoding for the lower 4 GB address space. Because EDK supports only the lower 4 GB space, there is normally no reason for users to use templates 1 through 3.

Table 3-14: **Bit Definitions for the TMPL_SEL_REG Register**

Bits	Field	Default	Description
0:31	SEL	`32'h3FFF_FFFF`	16 2-bit values for template register selection

DCRs for PLB Slave 0, SPLB 0 (0x34 to 0x43)

0x34: PLB Slave 0 Configuration Register (CFG_PLBS0), R/W

This register configures PLB Slave 0 operation (see Table 3-15). This register is initialized by embedded processor block attribute PPCS0_CONTROL.

Table 3-15: **Bit Definitions for the CFG_PLBS0 Register**

Bits	Field	Default	Description
0	LOCK_SESR	1	When this bit is set, the SESR and SEAR registers (`0x36`, `0x37`, and `0x38`) are locked.
1	Reserved	0	Reserved
2	DMA1_EN	0	• 0: Disable DMA1 • 1: Enable DMA1
3	DMA0_EN	`0`	• 0: Disable DMA0 • 1: Enable DMA0
4:5	DMA0_PRI	`00`	DMA0 priority • `00`: Lowest priority • `11`: Highest priority
6:7	DMA1_PRI	`00`	DMA1 priority • `00`: Lowest priority • `11`: Highest priority
8	Reserved	`0`	Reserved

Table 3-15: **Bit Definitions for the CFG_PLBS0 Register *(Cont'd)***

Bits	Field	Default	Description
9:11	THRMCI	`011`	Command translation for a read MCI threshold of 8. • `000`: Threshold of 1, a burst is turned into single transfers. • `001`: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • `010`: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • `011`: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • `100`: Threshold of 16, a burst is turned into burst-of-16 transfers, if applicable. • `101` to `111`: Reserved values that can cause unpredictable behaviors.
12	Reserved	`0`	Reserved
13:15	THRPLBM	`011`	Command translation for a read PLB Master threshold of 8. • `000`: Threshold of 1, a burst is turned into single transfers. • `001`: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • `010`: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • `011`: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • `100`: Threshold of 16, a burst is turned into burst-of-16 transfers, if applicable. • `101` to `111`: Reserved values that can cause unpredictable behaviors.
16	Reserved	`0`	Reserved
17:19	THWMCI	`011`	Command translation for a write MCI threshold of 8. • `000`: Threshold of 1, a burst is turned into single transfers. • `001`: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • `010`: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • `011`: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • `100`: Threshold of 16, a burst is turned into burst-of-16 transfers, if applicable. • `101` to `111`: Reserved values that can cause unpredictable behaviors.
20	Reserved	`0`	Reserved

Table 3-15: **Bit Definitions for the CFG_PLBS0 Register *(Cont'd)***

Bits	Field	Default	Description
21:23	THWPLBM	`011`	Command translation for a write PLB Master threshold of 8. • `000`: Threshold of 1, a burst is turned into single transfers. • `001`: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • `010`: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • `011`: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • `100`: Threshold of 16, a burst is turned into burst-of-16 transfers, if applicable. • `101` to `111`: Reserved values that can cause unpredictable behaviors.
24	Reserved	`0`	Reserved
25	LOCKXFER	`1`	Lock Transfers • 0: Disables lock transfers • 1: Enables lock transfers
26	RPIPE	`1`	Read Address Pipelining • 0: Disables read address pipelining • 1: Enables read address pipelining
27	WPIPE	`0`	Write Address Pipelining • 0: Disables write address pipelining • 1: Enables write address pipelining Cleared automatically when bit 28 is 0 to prevent posted write data.
28	WPOST	`1`	Write Posting • 0: No write posting (early data acknowledge) • 1: Enables write posting Bit 27 is cleared when this bit is 0. Only single transactions are supported when write posting is disabled. The interrupt status flag (bit 3 of Crossbar register `0x20`) is set if other types of transactions are received.
29	Reserved	`1`	Must be set to 1.
30	AERR_LOG	`0`	Log ABUS address mismatch error, when set (see bit 2 in register `0x39`)
31	CMD_CHK_DBL	`0`	Disable command (size) check, when set (see bits 0 and 1 in register `0x39`)

0x36: PLB Slave 0 Error Address Register (SEAR_U_PLBS0), Clear on Writes

This register is cleared by writing to register `0x38`. This register captures the upper 4-bit address of a 36-bit address of a failed transaction (see Table 3-16). The content is valid if bit 0 of register `0x38` is set. A failed transaction corresponds to a command address mismatch or an illegal command. This register is also used by the command sniffer (see registers `0x3D` and `0x3E`).

Table 3-16: **Bit Definitions for the SEAR_U_PLBS0 Register**

Bits	Field	Default	Description
0:27	Reserved	0	Reserved
28:31	U4BIT	0	Upper 4 bits of a 36-bit address

0x37: PLB Slave 0 Error Address Register (SEAR_L_PLBS0), Clear on Writes

This register is cleared by writing to register `0x38`. This register captures the lower 32-bit address of 36-bit address of a failed transaction (see Table 3-17). The content is valid if bit 0 of register `0x38` is set. A failed transaction corresponds to a command address mismatch or an illegal command. This register is also used by the command sniffer (see registers `0x3D` and `0x3E`).

Table 3-17: **Bit Definitions for the SEAR_L_PLBS0 Register**

Bits	Field	Default	Description
0:31	L32BIT	0	Lower 32 bits of a 36-bit address

0x38: PLB Slave 0 Error Status Register (SESR_PLBS0), Clear on Writes

This register captures the transaction qualifiers of a failed transaction (see Table 3-18). A failed transaction corresponds to a command address mismatch or an illegal command. The slave interface only supports the following commands:

- Single transfers
- 4, 8, and 16-word line transfers
- 32-bit, 64-bit, and 128-bit burst transfers

All other commands are considered illegal. Furthermore, if write posting is disabled, only single transfers are supported, any other types of transfers are considered illegal.

This register is also used by the command sniffer (see registers `0x3D` and `0x3E`).

The content is valid when bit 0 is set. See also registers `0x36` and `0x37`. This register is cleared by writing to it. When bit 0 of `0x34` is set, this register is only updated when bit 0 becomes 0. When bit 0 of `0x34` is not set, this register is updated every time an error or sniff event is detected.

Table 3-18: **Bit Definitions for the SESR_PLBS0 Register**

Bits	Field	Default	Description
0	VLD	`1'b0`	Valid • 0: No error detected • 1: Error or sniffed command detected
1	LOCKERR	`1'b0`	M_lockErr from the PLB Master
2:3	Reserved	`2'b0`	Reserved
4:5	MID	`2'b0`	Master ID
6:7	MSIZE	`2'b0`	Master Size
8:10	TYPE	`3'b0`	PLB Type. Only `000` for memory transfers is supported.

Table 3-18: **Bit Definitions for the SESR_PLBS0 Register *(Cont'd)***

Bits	Field	Default	Description
11	RNW	`1'b0`	Read/Write. • 0: Write • 1: Read
12:15	SIZE	`4'b0`	PLB Size
16:31	BE	`16'b0`	16-bit Byte Enable

0x39: PLB Slave 0 Miscellaneous Status Register (MISC_ST_PLBS0), Clear on Writes

This register contains miscellaneous status bits for PLB Slave 0 (see Table 3-19). Individual register bits are cleared by writing 1s to those bits that need to be cleared. Bit 3 of the Interrupt Status register is set if the configuration error bit, the illegal command bit, or the address mismatch error bit is set. Bit 7 of the Interrupt Status register is set if any FIFO overflow or underflow bit is set. None of these bits should ever be set under normal operating conditions.

Table 3-19: **Bit Definitions for the MISC_ST_PLBS0 Register**

Bits	Field	Default	Description
0	WPOST_CFG_ERR	0	When this bit is set, a write posting configuration error occurred. No write posting is configured (see register `0x34`) but a line or a burst transfer is detected.
1	ILLEGAL_CMD	0	Illegal command detected. The supported commands include: size = `4'h0`, `4'h1`, `4'h2`, `4'h3`, `4'hA`, `4'hB`, or `4'hC`. Qualified by bit 31 of register `0x34`.
2	ADDR_ERR	0	Address mismatch error. Qualified by bit 30 of register `0x34`.
3:17	Reserved	0	Reserved
18	FIFO_OF_RDAT	0	When set, a Read Data Queue overflow occurred
19	FIFO_UF_RDAT	0	When set, a Read Data Queue underflow occurred
20	FIFO_OF_WDAT	0	When set, a Write Data Queue overflow occurred
21	FIFO_UF_WDAT	0	When set, a Write Data Queue underflow occurred
22	FIFO_OF_SRDQ	0	When set, a Slave Read Queue overflow occurred
23	FIFO_UF_SRDQ	0	When set, a Slave Read Queue underflow occurred
24	FIFO_OF_SWRQ	0	When set, a Slave Write Queue overflow occurred
25	FIFO_UF_SWRQ	0	When set, a Slave Write Queue underflow occurred
26	FIFO_OF_MRDQ	0	When set, a Master Read Queue overflow occurred
27	FIFO_UF_MRDQ	0	When set, a Master Read Queue underflow occurred
28	FIFO_OF_MWRQ	0	When set, a Master Write Queue overflow occurred
29	FIFO_UF_MWRQ	0	When set, a Master Write Queue underflow occurred
30	FIFO_OF_INCMD	0	When set, an Input Command Queue overflow occurred
31	FIFO_UF_INCMD	0	When set, an Input Command Queue underflow occurred

0x3A: PLB Slave 0 PLB Error Status Register (PLBERR_ST_PLBS0), Clear on Writes

This register contains MIRQ status bits for PLB Slave 0 (see Table 3-20). Individual register bits are cleared by writing 1s to those bits that need to be cleared. Bits 28:31 are PLB MIRQ status bits, which can be set due to either the propagation of the slave MIRQ status or conversion of slave MwrErr into MIRQ because of write posting. Refer to the *PLB Architecture Specification* [Ref 4] for more information on the MIRQ signal.

Table 3-20: **Bit Definitions for the PLBERR_ST_PLBS0 Register**

Bits	Field	Default	Description
0:27	Reserved	0	Reserved
28	PLBS0_M0_MIRQ	0	PLB Slave 0, Master 0 MIRQ
29	PLBS0_M1_MIRQ	0	PLB Slave 0, Master 1 MIRQ
30	PLBS0_M2_MIRQ	0	PLB Slave 0, Master 2 MIRQ
31	PLBS0_M3_MIRQ	0	PLB Slave 0, Master 3 MIRQ

0x3B: PLB Slave 0 State Machine States Register (SM_ST_PLBS0), Read Only

This register indicates the states of the PLB Slave 0 state machine (see Table 3-21). This register is reserved for internal use.

Table 3-21: **Bit Definitions for the SM_ST_PLBS0 Register**

Bits	Field	Default	Description
0:31	Reserved	0	Reserved

0x3C: PLB Slave 0 Miscellaneous Control and Status Register (MISC_PLBS0), R/W, Write Only, or Read Only

This register contains miscellaneous control and status bits for PLB Slave 0 (see Table 3-22). Write-only bits always read as 0s.

Table 3-22: **Bit Definitions for the MISC_PLBS0 Register**

Bits	Field	Default	Type	Description
0	MODE_128N64	1	Read Only	• 0: PLBS0 is in 64-bit mode • 1: PLBS0 is in 128-bit mode
1:24	Reserved	0	-	Reserved
25	FIFO_RDAT_RST	0	Write Only	Write a 1 to this bit to reset the Read Data Queue
26	FIFO_WDAT_RST	0	Write Only	Write a 1 to this bit to reset the Write Data Queue
27	FIFO_SRDQ_RST	0	Write Only	Write a 1 to this bit to reset the Slave Read Queue
28	FIFO_SWRQ_RST	0	Write Only	Write a 1 to this bit to reset the Slave Write Queue
29	FIFO_MRDQ_RST	0	Write Only	Write a 1 to this bit to reset the Master Read Queue
30	FIFO_MWRQ_RST	0	Write Only	Write a 1 to this bit to reset the Master Write Queue
31	FIFO_INCMD_RST	0	Write Only	Write a 1 to this bit to reset the Input Command Queue

0x3D: PLB Slave 0 Command Sniffer Register (CMD_SNIFF_PLBS0), R/W

This register contains the description of a command (the address is specified in `0x3E`) that is to be monitored (see Table 3-23). The result is placed in registers `0x38` through `0x3A`. This register is used for debugging purposes.

Table 3-23: **Bit Definitions for the CMD_SNIFF_PLBS0 Register**

Bits	Field	Default	Description
0	ENABLE	0	• 0: Command capture is disabled • 1: Command capture is enabled
1:3	Reserved	000	Reserved
4:7	SIZE	0000	PLB size value to be matched
8	RNW	0	PLB RNW to be matched
9:11	MID	000	Master ID (0 – 4) to be matched
12	SPLBNDMA	0	• 0: Command from DMA engine to be matched • 1: Command from SPLB to be matched
13	Reserved	0	Reserved
14:15	SPLB_MID	00	FPGA logic master's ID (0 – 3) to be matched
16:17	SSIZE	00	SSIZE to be matched
18:24	Reserved	7'b0	Reserved
25	SIZE_EN	0	• 0: Disable size match • 1: Enable size match
26	RNW_EN	0	• 0: Disable RNW match • 1: Enable RNW match
27	MID_EN	0	• 0: Disable master ID match • 1: Enable master ID match
28	Reserved	0	Reserved
29	Reserved	0	Reserved
30	SSIZE_EN	0	• 0: Disable ssize match • 1: Enable ssize match
31	ADDR_EN	0	• 0: Disable address match • 1: Enable address match

0x3E: PLB Slave 0 Command Sniffer Address Register (CMD_SNIFFA_PLBS0), R/W

This register, used in conjunction with register `0x3D`, contains the address for command sniffing (see Table 3-24).

Table 3-24: **Bit Definitions for the CMD_SNIFFA_PLBS0 Register**

Bits	Field	Default	Description
0:31	L32BIT	0	Lower 32 bits of a 36-bit address

0x40 to 0x43: PLB Slave 0 Template Registers, R/W

Table 3-25 lists the set of four 32-bit template registers for PLB Slave 0. Selection of one of four registers for address mapping is done through the Template Selection Register. Each bit of a 32-bit register corresponds to a 128 MB address space for a 4 GB addressing. Set a bit to 1 to enable the corresponding 128 MB address space. These registers are initialized by embedded processor block attributes PPCS0_ADDRMAP_TMPL0 through PPCS0_ADDRMAP_TMPL3.

Table 3-25: **PLB Slave 0 Template Registers**

Address	Mnemonic	Default	Description
`0x40`	TMPL0_PLBS0_MAP	`32'hFFFF_FFFF`	Template Register 0 for PLB Slave 0
`0x41`	TMPL1_PLBS0_MAP	`32'hFFFF_FFFF`	Template Register 1 for PLB Slave 0
`0x42`	TMPL2_PLBS0_MAP	`32'hFFFF_FFFF`	Template Register 2 for PLB Slave 0
`0x43`	TMPL3_PLBS0_MAP	`32'hFFFF_FFFF`	Template Register 3 for PLB Slave 0

DCRs for PLB Slave 1, SPLB 1 (0x44 to 0x53)

0x44: PLB Slave 1 Configuration Register (CFG_PLBS1), R/W

This register configures PLB Slave 1 operation (see Table 3-26). This register is initialized by embedded processor block attribute PPCS1_CONTROL.

Table 3-26: **Bit Definitions for the CFG_PLBS1 Registers**

Bits	Field	Default	Description
0	LOCK_SESR	`1`	When this bit is set, the SESR and SEAR registers (`0x46`, `0x47`, and `0x48`) are locked.
1	Reserved	`0`	Reserved
2	DMA3_EN	`0`	• 0: Disable DMA3 • 1: Enable DMA3
3	DMA2_EN	`0`	• 0: Disable DMA2 • 1: Enable DMA2
4:5	DMA2_PRI	`00`	DMA2 priority • `00`: Lowest priority • `11`: Highest priority
6:7	DMA3_PRI	`00`	DMA3 priority • `00`: Lowest priority • `11`: Highest priority
8	Reserved	`0`	Reserved

Table 3-26: **Bit Definitions for the CFG_PLBS1 Registers *(Cont'd)***

Bits	Field	Default	Description
9:11	THRMCI	011	Command translation for a read MCI threshold of 8. • 000: Threshold of 1, a burst is turned into single transfers. • 001: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • 010: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • 011: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • 100: Threshold of 16, a burst of turned into burst-of-16 transfers, if applicable. • 101 to 111 are reserved values and can cause unpredictable behaviors.
12	Reserved	0	Reserved
13:15	THRPLBM	011	Command translation for a read PLB Master threshold of 8. • 000: Threshold of 1, a burst is turned into single transfers. • 001: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • 010: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • 011: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • 100: Threshold of 16, a burst of turned into burst-of-16 transfers, if applicable. • 101 to 111 are reserved values and can cause unpredictable behaviors.
16	Reserved	0	Reserved
17:19	THWMCI	011	Command translation for a write MCI threshold of 8. • 000: Threshold of 1, a burst is turned into single transfers. • 001: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • 010: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • 011: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • 100: Threshold of 16, a burst of turned into burst-of-16 transfers, if applicable. • 101 to 111 are reserved values and can cause unpredictable behaviors.
20	Reserved	0	Reserved

Table 3-26: **Bit Definitions for the CFG_PLBS1 Registers *(Cont'd)***

Bits	Field	Default	Description
21:23	THWPLBM	`011`	Command translation for a write PLB Master threshold of 8. • `000`: Threshold of 1, a burst is turned into single transfers. • `001`: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • `010`: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • `011`: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • `100`: Threshold of 16, a burst of turned into burst-of-16 transfers, if applicable. • `101` to `111` are reserved values and can cause unpredictable behaviors.
24	Reserved	`0`	Reserved
25	LOCKXFER	`1`	Lock Transfer. • 0: Disables lock transfers • 1: Enables lock transfers
26	RPIPE	`1`	Read Address Pipelining. • 0: Disables read address pipelining • 1: Enables read address pipelining
27	WPIPE	`0`	Write Address Pipelining. • 0: Disables write address pipelining • 1: Enables write address pipelining Cleared automatically if bit 28 is 0 to prevent posted write data.
28	WPOST	`1`	Write Posting. • 0: No write posting (early data ack) • 1: Enable write posting Bit 27 is cleared if this bit is 0. Only single transactions are supported if write posting is disabled. Interrupt status flag (bit 11 of Crossbar register `0x20`) is set if other types of transactions are received.
29	Reserved	`1`	Must be set to 1.
30	AERR_LOG	`0`	Log ABUS address mismatch error, if set (see bit 2, register `0x49`)
31	CMD_CHK_DBL	0	Disable command (size) check, if set (see bits 0 and 1, register `0x49`)

0x46: PLB Slave 1 Error Address Register (SEAR_U_PLBS1), Clear on Writes

This register is cleared by writing to register `0x48`. This register captures the upper 4-bit address of a 36-bit address of a failed transaction (see Table 3-27). The content is valid if bit 0 of register `0x48` is set. A failed transaction corresponds to a command address mismatch or an illegal command. This register is also used by the command sniffer (see registers `0x4D` and `0x4E`).

Table 3-27: **Bit Definitions for the SEAR_U_PLBS1 Register**

Bits	Field	Default	Description
0:27	Reserved	0	Reserved
28:31	U4BIT	0	Upper 4 bits of a 36-bit address

0x47: PLB Slave 1 Error Address Register (SEAR_L_PLBS1), Clear on Writes

This register is cleared by writing to register `0x48`. This register captures the lower 32-bit address of the 36-bit address of a failed transaction (see Table 3-28). This content is valid if bit 0 of register `0x48` is set. A failed transaction corresponds to a command address mismatch or an illegal command. This register is also used by the command sniffer (see registers `0x4D` and `0x4E`).

Table 3-28: **Bit Definitions for the SEAR_L_PLBS1 Register**

Bits	Field	Default	Description
0:31	L32BIT	0	Lower 32 bits of a 36-bit address

0x48: PLB Slave 1 Error Status Register (SESR_PLBS1), Clear on Writes

This register captures the transaction qualifiers of a failed transaction (see Table 3-29). A failed transaction corresponds to a command address mismatch or an illegal command. The slave interface only supports the following commands:

- Single transfers
- 4, 8, and 16-word line transfers
- 32-bit, 64-bit, and 128-bit burst transfers

All other commands are considered illegal. Furthermore, if write posting is disabled, only single transfers are supported, and any other types of transfers are considered illegal.

This register is also used by the command sniffer (see registers `0x4D` and `0x4E`).

The content is valid if bit 0 is set. See also registers `0x46` and `0x47`. This register is cleared by writing to it. If bit 0 of register `0x44` is set, this register is only updated when bit 0 becomes 0. If bit 0 of register `0x44` is not set, this register is updated every time an error or sniff event is detected.

Table 3-29: **Bit Definitions for the SESR_PLBS1 Register**

Bits	Field	Default	Description
0	VLD	`1'b0`	Valid • 0: No error detected • 1: Error or sniffed command detected
1	LOCKERR	`1'b0`	M_lockErr from the PLB Master
2:3	Reserved	`2'b0`	Reserved
4:5	MID	`2'b0`	Master ID
6:7	MSIZE	`2'b0`	Master Size
8:10	TYPE	`3'b0`	PLB Type. Only `000` for memory transfers is supported.

Table 3-29: **Bit Definitions for the SESR_PLBS1 Register *(Cont'd)***

Bits	Field	Default	Description
11	RNW	`1'b0`	Read/Write. • 0: Write • 1: Read
12:15	SIZE	`4'b0`	PLB Size
16:31	BE	`16'b0`	16-bit Byte Enable

0x49: PLB Slave 1 Miscellaneous Status Register (MISC_ST_PLBS1), Clear on Writes

This register contains miscellaneous status bits for PLB Slave 1 (see Table 3-30). Individual register bits are cleared by writing 1s to those bits that need to be cleared. Bit 11 of the Interrupt Status register is set if the configuration error bit, the illegal command bit, or the address mismatch error bit is set. Bit 15 of the Interrupt Status register is set if any FIFO overflow or underflow bit is set. None of these bits should ever be set under normal operating conditions.

Table 3-30: **Bit Definitions for the MISC_ST_PLBS1 Register**

Bits	Field	Default	Description
0	WPOST_CFG_ERR	0	When this bit is set, a write posting configuration error occurred. No write posting is configured (see register `0x44`) but a line or a burst transfer is detected.
1	ILLEGAL_CMD	0	Illegal command detected. The supported commands include: size = `4'h0`, `4'h1`, `4'h2`, `4'h3`, `4'hA`, `4'hB`, or `4'hC`. Qualified by bit 31 of register `0x44`.
2	ADDR_ERR	0	Address mismatch error. Qualified by bit 30 of register `0x44`.
3:17	Reserved	0	Reserved
18	FIFO_OF_RDAT	0	When this bit is set, a Read Data Queue overflow occurred
19	FIFO_UF_RDAT	0	When this bit is set, a Read Data Queue underflow occurred
20	FIFO_OF_WDAT	0	When this bit is set, a Write Data Queue overflow occurred
21	FIFO_UF_WDAT	0	When this bit is set, a Write Data Queue underflow occurred
22	FIFO_OF_SRDQ	0	When this bit is set, a Slave Read Queue overflow occurred
23	FIFO_UF_SRDQ	0	When this bit is set, a Slave Read Queue underflow occurred
24	FIFO_OF_SWRQ	0	When this bit is set, a Slave Write Queue overflow occurred

Table 3-30: **Bit Definitions for the MISC_ST_PLBS1 Register *(Cont'd)***

Bits	Field	Default	Description
25	FIFO_UF_SWRQ	0	When this bit is set, a Slave Write Queue underflow occurred
26	FIFO_OF_MRDQ	0	When this bit is set, a Master Read Queue overflow occurred
27	FIFO_UF_MRDQ	0	When this bit is set, a Master Read Queue underflow occurred
28	FIFO_OF_MWRQ	0	When this bit is set, a Master Write Queue overflow occurred
29	FIFO_UF_MWRQ	0	When this bit is set, a Master Write Queue underflow occurred
30	FIFO_OF_INCMD	0	When this bit is set, an Input Command Queue overflow occurred
31	FIFO_UF_INCMD	0	When this bit is set, an Input Command Queue underflow occurred

0x4A: PLB Slave 1 PLB Error Status Register (PLBERR_ST_PLBS1), Clear on Writes

This register contains the MIRQ status bits for PLB Slave 1 (see Table 3-31). Individual register bits are cleared by writing 1s to those bits that need to be cleared. Bits 28:31 are PLB MIRQ status bits, which can be set due to either the propagation of the slave MIRQ status or conversion of slave MwrErr into MIRQ because of write posting.

Table 3-31: **Bit Definitions for the PLBERR_ST_PLBS1 Register**

Bits	Field	Default	Description
0:27	Reserved	0	Reserved
28	PLBS1_M0_MIRQ	0	PLB Slave 1, Master 0 MIRQ
29	PLBS1_M1_MIRQ	0	PLB Slave 1, Master 1 MIRQ
30	PLBS1_M2_MIRQ	0	PLB Slave 1, Master 2 MIRQ
31	PLBS1_M3_MIRQ	0	PLB Slave 1, Master 3 MIRQ

0x4B: PLB Slave 1 State Machine States Register (SM_ST_PLBS1), Read Only

This register indicates the states of the state machine for PLB Slave 1 (see Table 3-32). This register is reserved for internal use.

Table 3-32: **Bit Definitions for the SM_ST_PLBS1 Register**

Bits	Field	Default	Description
0:31	Reserved	0	Reserved

0x4C: PLB Slave 1 Miscellaneous Control and Status Register (MISC_PLBS1), R/W, Write Only, or Read Only

This register contains miscellaneous control and status bits for PLB Slave 1 (see Table 3-33). Write-only bits always read as 0s.

Table 3-33: **Bit Definitions for the MISC_PLBS1 Register**

Bits	Field	Default	Type	Description
0	MODE_128N64	1	Read Only	• 0: PLBS1 is in 64-bit mode • 1: PLBS1 is in 128-bit mode
1:24	Reserved	0	-	Reserved
25	FIFO_RDAT_RST	0	Write Only	Write a 1 to this bit to reset the Read Data Queue
26	FIFO_WDAT_RST	0	Write Only	Write a 1 to this bit to reset the Write Data Queue
27	FIFO_SRDQ_RST	0	Write Only	Write a 1 to this bit to reset the Slave Read Queue
28	FIFO_SWRQ_RST	0	Write Only	Write a 1 to this bit to reset the Slave Write Queue
29	FIFO_MRDQ_RST	0	Write Only	Write a 1 to this bit to reset the Master Read Queue
30	FIFO_MWRQ_RST	0	Write Only	Write a 1 to this bit to reset the Master Write Queue
31	FIFO_INCMD_RST	0	Write Only	Write a 1 to this bit to reset the Input Command Queue

0x4D: PLB Slave 1 Command Sniffer Register (CMD_SNIFF_PLBS1), R/W

This register contains the description of a command (whose address is specified in register `0x4E`) that is to be monitored (see Table 3-34). The results are placed in registers `0x48` through `0x4A`. This register is used for debugging purposes.

Table 3-34: **Bit Definitions for the CMD_SNIFF_PLBS1 Register**

Bits	Field	Default	Description
0	ENABLE	0	• 0: Command capture disabled • 1: Command command enabled
1:3	Reserved	000	Reserved
4:7	SIZE	0000	PLB size value to be matched
8	RNW	0	PLB RNW to be matched
9:11	MID	000	Master ID (0 – 4) to be matched
12	SPLBNDMA	0	• 0: Command from DMA engine to be matched • 1: Command from SPLB to be matched
13	Reserved	0	Reserved
14:15	SPLB_MID	00	FPGA logic master's ID (0 – 3) to be matched

Table 3-34: **Bit Definitions for the CMD_SNIFF_PLBS1 Register *(Cont'd)***

Bits	Field	Default	Description
16:17	SSIZE	`00`	SSIZE to be matched
18:24	Reserved	`7'h0`	Reserved
25	SIZE_EN	0	• 0: Disable size match • 1: Enable size match
26	RNW_EN	0	• 0: Disable RNW match • 1: Enable RNW match
27	MID_EN	0	Enable master ID match, if set • 0: Disable master ID match • 1: Enable master ID match
28	Reserved	0	Reserved
29	Reserved	0	Reserved
30	SSIZE_EN	0	• 0: Disable ssize match • 1: Enable ssize match
31	ADDR_EN	0	• 0: Disable address match • 1: Enable address match

0x4E: PLB Slave 1 Command Sniffer Address (CMD_SNIFFA_PLBS1), R/W

This register, used in conjunction with register `0x4D`, contains the address (lower 32 bits) for command sniffing (see Table 3-35).

Table 3-35: **Bit Definitions for the CMD_SNIFFA_PLBS1 Register**

Bits	Field	Default	Description
0:31	L32BIT	0	Lower 32-bit of a 36-bit address

0x50 to 0x53: PLB Slave 1 Template Registers, R/W

Table 3-36 lists the set of four 32-bit template registers for PLB Slave 1. Selection of one of four registers for address mapping is done through the Template Selection Register. Each bit of a 32-bit register corresponds to 128 MB address space for a total of 4 GB addressing. Set a bit to 1 to enable the corresponding 128 MB address space. These registers are initialized by embedded processor block attributes PPCS1_ADDRMAP_TMPL0 through PPCS1_ADDRMAP_TMPL3.

Table 3-36: **PLB Slave 1 Template Registers**

Address	Mnemonic	Default	Description
`0x50`	TMPL0_PLBS1_MAP	`32'hFFFF_FFFF`	Template Register 0 for PLB Slave 1
`0x51`	TMPL1_PLBS1_MAP	`32'hFFFF_FFFF`	Template Register 1 for PLB Slave 1
`0x52`	TMPL2_PLBS1_MAP	`32'hFFFF_FFFF`	Template Register 2 for PLB Slave 1
`0x53`	TMPL3_PLBS1_MAP	`32'hFFFF_FFFF`	Template Register 3 for PLB Slave 1

DCRs for PLB Master, MPLB (0x54 to 0x5F)

0x54: PLB Master Configuration Register (CFG_PLBM), R/W

This register configures PLB Master operation (see Table 3-37). This register is initialized by embedded processor block attribute PPCM_CONTROL.

Table 3-37: **Bit Definitions for the CFG_PLBM Register**

Bits	Field	Default	Description
0	LOCK_SESR	`1`	When this bit is set, the SESR and SEAR registers (`0x56`, `0x57`, and `0x58`) are locked.
1:22	Reserved	`0`	Reserved
23	Reserved	`0`	Must be set to 0.
24	XBAR_PRIORITY_ENA	`1`	• 0: Priority is disabled during crossbar arbitration • 1: Priority is enabled during crossbar arbitration
25	Reserved	`0`	Reserved (can lead to unexpected behavior, if set to 1).
26	SL_ETERM_MODE	`0`	When this bit is set, slave early burst termination is supported. Bits 28 and 29 are cleared automatically when this bit is set. This mode prevents R/W command re-ordering.
27	LOCKXFER	`1`	Lock Transfers. • 0: Disables lock transfers • 1: Enables lock transfers
28	RPIPE	`1`	Read Address Pipelining. • 0: Disables read address pipelining • 1: Enables read address pipelining Can be cleared directly or through bit 26.
29	WPIPE	`1`	Write Address Pipelining. • 0: Disables write address pipelining • 1: Enables write address pipelining Can be cleared directly or through bit 26. This bit is cleared if bit 30 is 0 to prevent posted write data.
30	WPOST	`1`	Write Posting. • 0: No write posting (early data ack) • 1: Enable write posting Bit 29 is cleared if this bit is 0. Only single transactions are supported if write posting is disabled. Interrupt status flag (bit 17 of Crossbar register `0x20`) is set if other types of transactions are received.
31	Reserved	`1`	Must be set to 1.

0x56: FPGA Logic Slave Error Address Register (FSEAR_U_PLBM), Clear on Writes

This register is cleared by writing to register `0x58`. This register captures the upper 4-bit address of a 36-bit address of a failed transaction (see Table 3-38). The content is valid if bit 0 of register `0x58` is set. A failed transaction corresponds to one of the following conditions: address time-out, data time-out, PLB slave MwrErr, or PLB slave MrdErr. This register is also used by the command sniffer (see registers `0x5D` – `0x5E`).

Table 3-38: **Bit Definitions for the FSEAR_U_PLBM Register**

Bits	Field	Default	Description
0:27	Reserved	0	Reserved
28:31	U4BIT	0	Upper 4 bits of a 36-bit address

0x57: FPGA Logic Slave Error Address Register (FSEAR_L_PLBM), Clear on Writes

This register is cleared by writing to register `0x58`. This register captures the lower 32-bit address of a 36-bit address of a failed transaction (see Table 3-39). The content is valid if bit 0 of register `0x58` is set. A failed transaction corresponds to one of the following conditions: address time-out, data time-out, PLB slave MwrErr, or PLB slave MrdErr. This register is also used by the command sniffer (see registers `0x5D` – `0x5E`).

Table 3-39: **Bit Definitions for the FSEAR_L_PLBM Register**

Bits	Field	Default	Description
0:31	L32BIT	0	Lower 32 bits of a 36-bit address

0x58: FPGA Logic Slave Error Status Register (FSESR_PLBM), Clear on Writes

This register is cleared by writing to it. This register captures the transaction qualifiers of a failed transaction (see Table 3-40). A failed transaction corresponds to one of the following conditions: address time-out, data time-out, PLB slave MwrErr, or PLB slave MrdErr.

This register is also used by the command sniffer (see registers `0x5D` – `0x5E`).

The content is valid if bit 0 is set. See also registers `0x56` and `0x57`. If bit 0 of register `0x54` is set, this register is only updated when bit 0 becomes 0. If bit 0 of register `0x54` is not set, the register is updated every time an error or sniff event is detected.

Table 3-40: **Bit Definitions for the FSESR_PLBM Register**

Bits	Field	Default	Description
0	VLD	0	Valid • 0: No error detected • 1: Error or sniffed command detected
1	LOCKERR	0	M_lockErr to the PLB slave
2	PLBS_DMA	0	• 1: Command from PLB slave 0 or 1 • 0: Command from a DMA engine. Value is only valid if MID is 3 or 4.

Table 3-40: **Bit Definitions for the FSESR_PLBM Register *(Cont'd)***

Bits	Field	Default	Description
3:5	MID	`3'b0`	Master ID. • `000`: ICUR • `001`: DCUW • `010`: DCUR • `011`: PLBS0 • `100`: PLBS1
6:7	SSIZE	`2'b0`	Slave size (`00`, `01`, or `10`). `11` indicates address time-out.
8:10	TYPE	`3'b0`	PLB Type. Only `000` for memory transfers is supported.
11	RNW	0	Read/Write. • 0: Write • 1: Read
12:15	SIZE	`4'b0`	PLB Size
16:31	BE	`16'b0`	16-bit Byte Enable

0x59: PLB Master Miscellaneous Status Register (MISC_ST_PLBM), Clear on Writes

This register contains miscellaneous status bits for the PLB master (see Table 3-41). Individual register bits are cleared by writing 1s to those bits that need to be cleared. Bit 17 of the Interrupt Status register is set if either or both of the configuration error bits are set. Bit 24 of the Interrupt Status register is set if any FIFO overflow or underflow bit is set. None of these bits should ever be set under normal operating conditions.

Table 3-41: **Bit Definitions for the MISC_ST_PLBM Register**

Bits	Field	Default	Description
0	WPOST_CFG_ERR	0	When this bit is set, a write posting configuration error occurred. No write posting is configured (see register `0x54`) but a line or a burst transfer is detected.
1	ETERM_CFG_ERR	0	When this bit is set, a slave early burst termination configuration error occurred. Early termination is not configured (see register `0x54`), but early termination is detected.
2:17	Reserved	0	Reserved
18	FIFO_OF_RDAT	0	When set, a Read Data Queue overflow occurred
19	FIFO_UF_RDAT	0	When set, a Read Data Queue underflow occurred
20	FIFO_OF_WDAT	0	When set, a Write Data Queue overflow occurred
21	FIFO_UF_WDAT	0	When set, a Write Data Queue underflow occurred
22	FIFO_OF_SRDQ	0	When set, a Slave Read Queue overflow occurred
23	FIFO_UF_SRDQ	0	When set, a Slave Read Queue underflow occurred
24	FIFO_OF_SWRQ	0	When set, a Slave Write Queue overflow occurred
25	FIFO_UF_SWRQ	0	When set, a Slave Write Queue underflow occurred

Table 3-41: **Bit Definitions for the MISC_ST_PLBM Register *(Cont'd)***

Bits	Field	Default	Description
26	FIFO_OF_MRDQ	0	When set, a Master Read Queue overflow occurred
27	FIFO_UF_MRDQ	0	When set, a Master Read Queue underflow occurred
28	FIFO_OF_MWRQ	0	When set, a Master Write Queue overflow occurred
29	FIFO_UF_MWRQ	0	When set, a Master Write Queue underflow occurred
30	FIFO_OF_INCMD	0	When set, an Input Command Queue overflow occurred
31	FIFO_UF_INCMD	0	When set, an Input Command Queue underflow occurred

0x5A: PLB Master PLB Error Status Register (PLBERR_ST_PLBM), Read Only or Clear on Writes

This register contains MIRQ status bits for the PLB master (see Table 3-42). Bits 19:31 are PLB MIRQ status bits that can be set due to either the propagation of the slave MIRQ status or conversion of slave MwrErr into MIRQ because of write posting.

- If the slave PLB MIRQ signal, which is latched at the slave, is set, all MIRQ bits in the register are set. They are read only, so they are cleared when the PLB MIRQ is cleared.
- If the slave PLB MwrErr, which is a pulse, is set and write posting is enabled, one of the MIRQ bits is set. In this case, the register bit is cleared by writing a 1 to the bit.

Bits 19 and 20 correspond to MIRQ errors for writes that originated from the DMA engines in SPLB0/1.

Table 3-42: **Bit Definitions for the PLBERR_ST_PLBM Register**

Bits	Field	Default	Description
0:18	Reserved	0	Reserved
19	PLBS0_DMA_MIRQ	0	PLB Slave 0, DMA MIRQ
20	PLBS1_DMA_MIRQ	0	PLB Slave 1, DMA MIRQ
21	C440_MIRQ_ICUR	0	Processor ICUR MIRQ
22	C440_MIRQ_DCUW	0	Processor DCUW MIRQ
23	C440_MIRQ_DCUR	0	Processor DCUR MIRQ
24	PLBS0_M0_MIRQ	0	PLB Slave 0, Master 0 MIRQ
25	PLBS0_M1_MIRQ	0	PLB Slave 0, Master 1 MIRQ
26	PLBS0_M2_MIRQ	0	PLB Slave 0, Master 2 MIRQ
27	PLBS0_M3_MIRQ	0	PLB Slave 0, Master 3 MIRQ
28	PLBS1_M0_MIRQ	0	PLB Slave 1, Master 0 MIRQ
29	PLBS1_M1_MIRQ	0	PLB Slave 1, Master 1 MIRQ
30	PLBS1_M2_MIRQ	0	PLB Slave 1, Master 2 MIRQ
31	PLBS1_M3_MIRQ	0	PLB Slave 1, Master 3 MIRQ

0x5B: PLB Master State Machine States Register (SM_ST_PLBM), Read Only

This register indicates the states of the state machine for the PLB Master (see Table 3-43). This register is reserved.

Table 3-43: **Bit Definitions for the SM_ST_PLBM Register**

Bits	Field	Default	Description
0:31	Reserved	0	Reserved

0x5C: PLB Master Miscellaneous Control and Status Register (MISC_PLBM), R/W or Write Only

This register contains miscellaneous control and status bits for the PLB Master (see Table 3-44). Write-only bits always read as 0s.

Table 3-44: **Bit Definitions for the MISC_PLBM Register**

Bits	Field	Default	Type	Description
0	Reserved	0	Write Only	Reserved
1:2	FLUSH_MODE	`00`	R/W	Flush mode select • `00`: Automatic addrAck time-out flush • `01` - `10`: Reserved • `11`: No flush
3	Reserved	0	R/W	Reserved
4:24	Reserved	0	-	Reserved
25	FIFO_RDAT_RST	0	Write Only	Write a 1 to this bit to reset the Read Data Queue
26	FIFO_WDAT_RST	0	Write Only	Write a 1 to this bit to reset the Write Data Queue
27	FIFO_SRDQ_RST	0	Write Only	Write a 1 to this bit to reset the Slave Read Queue
28	FIFO_SWRQ_RST	0	Write Only	Write a 1 to this bit to reset the Slave Write Queue
29	FIFO_MRDQ_RST	0	Write Only	Write a 1 to this bit to reset the Master Read Queue
30	FIFO_MWRQ_RST	0	Write Only	Write a 1 to this bit to reset the Master Write Queue
31	FIFO_INCMD_RST	0	Write Only	Write a 1 to this bit to reset the Input Command Queue

0x5D: PLB Master Command Sniffer Register (CMD_SNIFF_PLBM), R/W

This register contains the description of a command (whose address is specified in register `0x5E`) that is to be monitored (see Table 3-45). The results are placed in registers `0x58` through `0x5A`. This register is used for debugging purposes.

Table 3-45: **Bit Definitions for the CMD_SNIFF_PLBM Register**

Bits	Field	Default	Description
0	ENABLE	0	• 0: Command capture is disabled • 1: Command capture is enabled
1:3	Reserved	`000`	Reserved
4:7	SIZE	`0000`	PLB size value to be matched
8	RNW	0	PLB RNW to be matched

Table 3-45: **Bit Definitions for the CMD_SNIFF_PLBM Register *(Cont'd)***

Bits	Field	Default	Description
9:11	MID	000	Master ID (0 – 4) to be matched
12	SPLBNDMA	0	• 0: Command from DMA engine to be matched • 1: Command from SPLB to be matched
13	Reserved	0	Reserved
14:15	SPLB_MID	00	FPGA logic master's ID (0 – 3) to be matched
16:17	SSIZE	00	SSIZE to be matched
18:24	Reserved	7'h0	Reserved
25	SIZE_EN	0	• 0: Disable size match • 1: Enable size match
26	RNW_EN	0	• 0: Disable RNW match • 1: Enable RNW match
27	MID_EN	0	• 0: Disable master ID match • 1: Enable master ID match
28	SPLBNDMA_EN	0	• 0: Disable SPLBndma match • 1: Enable SPLBndma match
29	SPLB_MID_EN	0	• 0: Disable SPLB_MID match • 1: Enable SPLB_MID match
30	SSIZE_EN	0	• 0: Disable ssize match • 1: Enable ssize match
31	ADDR_EN	0	• 0: Disable address match • 1: Enable address match

0x5E: PLB Master Command Sniffer Address (CMD_SNIFFA_PLBM), R/W

This register, used in conjunction with register `0x5D`, contains the ABUS address (lower 32 bits) for command sniffing (see Table 3-46).

Table 3-46: **Bit Definitions for the CMD_SNIFFA_PLBM Register**

Bits	Field	Default	Description
0:31	L32BIT	0	Lower 32 bits of a 36-bit address

Chapter 4

PLB Interface

MPLB Interface

The primary purpose of the crossbar MPLB interface is to provide access from the processor to PLB-based memory (if any) and non-memory peripherals. The MPLB also allows DMA access from processor block LocalLink interfaces to PLB-based memory (if any). The MPLB also allows access from PLB-based masters outside the embedded processor block in Virtex-5 FXT FPGAs, connected via one of the SPLB interfaces, to PLB-based memories and non-memory peripherals, which are also to be shared with the processor. See "PLB Interconnection Techniques," page 124.

The crossbar MPLB interface is a 128-bit wide master on the PLB. The PLB to which it is connected can be populated by slaves with any mixture of data widths and functional capabilities. However, care should be taken that burst and cache-line transfer requests are not directed at slave peripherals that do not support burst or cache-line transfers. The MPLB interface does not automatically translate burst or cache-line transfers into single-unit transfers to satisfy the limitations of such slaves. Therefore, all pages in the processor's MMU that contain addresses of such slaves should be designated as cache-inhibited.

Transaction Types

The MPLB can request the following transaction types allowed under the *PLB Architecture Specification* [Ref 4].

- Single Unit

  The MPLB can produce single-unit transfers of unaligned data from 1 to 16 bytes. These can be requested by the processor's data load/store unit when accessing cache-inhibited spaces, or can originate from PLB masters connected to the SPLB. They also can be automatically generated at the head and/or tail of burst transfers, originating at the crossbar SPLB or DMA (LocalLink) interfaces, that begin and/or end on unaligned addresses. All single-unit transfers of any size that can be issued by the MPLB interface are compatible with all slaves of any size and capability range.

- Bursts

  The MPLB can produce fixed length bursts of 2 to 16 quadwords. The actual number of data beats used to transfer the entire burst depends on the size of the targeted slave (up to 32 beats for 64-bit slaves, and up to 64 beats for 32-bit slaves). Burst transfers typically originate at the crossbar's SPLB or DMA interfaces. (The processor's instruction fetch and data load/store units never request burst-type transfers.) The crossbar can be configured to limit the maximum size, in quadwords, of all burst transfers through the crossbar to be 16, 8 (default), 4, or 2. Any burst request received by the crossbar that exceeds the configured limit is broken into multiple bursts of up to the maximum size. Bursts produced by the MPLB interface are compatible with all slaves that support burst transfers, provided that any data FIFO in the slave required to buffer read or write data is large enough to handle the maximum burst length that the crossbar is configured to produce.

- Cache Lines

  The MPLB can produce transfers of aligned 8-word (32-byte) cache lines. Typically, only the processor requests cache-line transfers. The processor instruction fetch unit requests only 8-word cache-line transfers. The processor data load/store unit can request either 8-word (typically) or 4-word cache-line transfers. Cache-line requests received by the crossbar SPLB interface are also propagated through the crossbar. The targeted slave must return cache-line reads in an aligned sequential order. Cache-line reads can be returned beginning with either the first word of the cache line or the quadword containing the target word, and must always proceed sequentially through the remainder of the cache line, incrementing according to the size of the data units being transferred, as shown in Table 4-1. Cache-line writes are always aligned to the beginning of the cache line. All cache-line transfers produced by the MPLB interface are compatible with all slaves that support cache-line transfers.

  The MPLB converts 4-word cache lines into single transfers, so that the MPLB does not send out 4-word cache line commands to PLB slaves. Furthermore, the MPLB automatically adjusts the cache-line read address to a quadword boundary.

  In the example in Table 4-1, the processor requests an 8-word cache-line read from a 64-bit slave on the MPLB interface. The least-significant five bits of the physical address requested by the processor (Abus[27:31]) are "`11100`", meaning that the *target word* is the seventh word of the cache line. The embedded processor block crossbar always adjusts the target address so that it is quadword aligned. Therefore, in the transaction requested on the MPLB interface, Abus[27:31] = "`10000`". The target word originally requested by the processor is shaded in the table.

Table 4-1: **Allowable Cache-Line Read Data Ordering on the MPLB**

Data Beat	Doubleword Returned by Slave (Least-Significant Five Address Bits)							
	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Alternative 1: Cache-Line Aligned								
1	00000	00001	00010	00011	00100	00101	00110	00111
2	01000	01001	01010	01011	01100	01101	01110	01111
3	10000	10001	10010	10011	10100	10101	10110	10111
4	11000	11001	11010	11011	11100	11101	11110	11111
Alternative 2: Target Word First								
1	10000	10001	10010	10011	10100	10101	10110	10111
2	11000	11001	11010	11011	11100	11101	11110	11111
3	00000	00001	00010	00011	00100	00101	00110	00111
4	01000	01001	01010	01011	01100	01101	01110	01111

MPLB Interface Features

The MPLB can operate at an integer 1:N clock ratio with respect to the crossbar interconnect clock (CPMINTERCONNECTCLK), where $1 \leq N \leq 16$.

The MPLB never prematurely terminates a burst. If the originating PLB master prematurely terminates a burst (the processor never generates burst type transfers), the command translation process in the SPLB converts that to a fixed-length burst or a single or a mix of fixed-length bursts and single transfers.

When a soft PLB slave does not respond to an MPLB transfer request, the soft arbiter sends the time-out signal to the MPLB, causing the command to be flushed. For a time-out write command, the MPLB flushes the command and the associated write data, and a write error signal or MIRQ propagates back to the originating master. For a time-out read command, the MPLB flushes the command and generates dummy read data back to the originating master with the read error signal asserted.

For all transactions targeting narrower slaves (32- or 64-bit native data width), the MPLB performs the required data steering to accept (read) or present (write) data in the byte lanes expected by the slave. For single-unit transfers targeting narrower slaves in which the number of enabled byte lanes exceeds the native width of the slave (as indicated by PLB_MSsize), the MPLB automatically generates the required conversion cycles (issues subsequent bus requests) to complete the originally requested transfer.

The MPLB fully supports address pipelining. If acknowledged, the MPLB continues to issue up to four read requests and four write requests onto the PLB, as queued up by the crossbar. The MPLB interface can be configured to disable address pipelining.

The MPLB supports overlapped read and write transfers. In other words, a read request that is acknowledged while a prior write data transfer is in progress can result in read data being transferred over the bus concurrently with write data and vice-versa.

The MPLB supports a *Sync* control signal, which is specific to the crossbar. The Sync signal is mapped to the PLB signal TAttribute[7], which is set by the processor when Storage Attribute “U3” is set in the TLB for the page addressed by the transfer. When asserted by any master targeting the MPLB (and provided the MPLB is configured to enable the Sync feature), the Sync signal causes the crossbar to block subsequent MPLB transfers until the target slave deasserts its MBusy signal back to the MPLB master. Also, the crossbar propagates the slave's MBusy signal back to the originating master until deasserted by the slave. This allows masters to ensure that the transaction (such as a posted write) is actually completed by the slave before issuing subsequent transfers. Without the Sync signal, MBusy is normally deasserted to the originating master as soon as the transfer completes on the MPLB.

MPLB Interface Signals

Table 4-2 summarizes the MPLB Interface signals in alphabetical order.

Table 4-2: **MPLB Interface Signals**

Port Name	Connects to PLB Signal	Dir	Default Value	Description
CPMPPCMPLBCLK	PLB_Clk	I	1	This clock synchronizes the MPLB interface to the connected PLB arbiter. The MPLB can operate at an integer 1:N clock ratio with respect to the crossbar interconnect clock (CPMINTERCONNECTCLK), where 1 ≤N ≤16.
PLBPPCMADDRACK	PLB_MAddrAck	I	0	The slave asserts this signal to indicate that it has acknowledged the address and will latch the address and all transfer qualifiers at the end of the current clock cycle.
PLBPPCMMBUSY	PLB_MBusy	I	0	The slave asserts this signal to indicate that the slave is busy performing a read or a write transfer or has a read or write transfer pending that was initiated by the MPLB interface. This signal is propagated back to the originating master until the slave deasserts it (if the Sync signal was asserted at the time of the request) or until the current transfer completes (if Sync was not asserted; see M_TAttribute[7]).
PLBPPCMMIRQ	PLB_MIRQ	I	0	The slave asserts this signal to indicate that it encountered an event that it has deemed important to the master. MIRQ is generally used to indicate an error condition that is not associated with a read or write transfer currently in progress. Propagated back to the originating master, if known; otherwise broadcast to all PLB masters connected to the crossbar (including the processor).
PLBPPCMMRDERR	PLB_MRdErr	I	0	The slave asserts this signal to indicate that it encountered an error during a read transfer that was initiated by this master, typically a transfer still in progress. Propagated back to the originating master.
PLBPPCMMWRERR	PLB_MWrErr	I	0	The slave drives this input to indicate that it encountered an error during a write transfer that was initiated by this master, typically a transfer still in progress. Propagated back to the originating master if the transfer is still in progress (not posted); otherwise MIRQ is sent back to the master.

Table 4-2: **MPLB Interface Signals *(Cont'd)***

Port Name	Connects to PLB Signal	Dir	Default Value	Description
PLBPPCMRDBTERM	PLB_MRdBTerm	I	0	The slave asserts this signal to indicate to a master that the current burst read transfer in progress is to be terminated following the next read data beat.
PLBPPCMRDDACK	PLB_MRdDAck	I	0	The slave asserts this signal during a read transfer to indicate that the data on the Sl_rdDBus bus is valid and must be latched at the end of the current clock cycle.
PLBPPCMRDDBUS[0:127]	PLB_MRdDBus [0:127]	I	x	128-bit data bus used to transfer data from a slave to a master during a PLB read transfer.
PLBPPCMRDPENDPRI[0:1]	PLB_rdPendPri[0:1]	I	x	Not used by the embedded processor block in Virtex-5 FPGAs. The PLB arbiter drives these signals, which are valid when the PLB_rdPendReq signal is asserted. These signals indicate the highest priority of any active read request input from all masters attached to the PLB or a pipelined read transfer that has been acknowledged and is pending.
PLBPPCMRDPENDREQ	PLB_rdPendReq	I	0	Not used by the embedded processor block in Virtex-5 FPGAs. The PLB arbiter asserts this signal to indicate that a master has a read request pending on the PLB or a secondary read transfer has been acknowledged and is pending.
PLBPPCMRDWDADDR[0:3]	PLB_MRdWdAddr[0:3]	I	0	The slave drives this bus to indicate the relative word address within the cache line of the first word of the data unit currently being transferred as part of a requested cache-line read transfer. Slaves are required to respond to line reads by returning data in an aligned sequential order. Cache-line reads may be returned beginning either with the first word of the cache line or the data unit (word, doubleword or quadword aligned, depending on the slave's native data width) containing the target word. These reads must always proceed sequentially through the remainder of the cache line, incrementing according to the size of the data units being transferred.

Table 4-2: **MPLB Interface Signals *(Cont'd)***

Port Name	Connects to PLB Signal	Dir	Default Value	Description
PLBPPCMREARBITRATE	PLB_MRearbitrate	I	0	The slave asserts this signal to indicate that it cannot perform the currently requested transfer at this time, and it requires the master to temporarily deassert its request and the PLB arbiter to re-arbitrate the bus. The crossbar MPLB master always subsequently re-requests the same command. It does not issue queued commands out of sequence.
PLBPPCMREQPRI[0:1]	PLB_reqPri[0:1]	I	x	Not used by the embedded processor block in Virtex-5 FPGAs. The PLB arbiter drives these signals, which are valid any time the PLB_rdPendReq or PLB_wrPendReq signals are asserted. These signals indicate the priority of the current request that the PLB arbiter has granted and is gating to the slaves.
PLBPPCMSSIZE[0:1]	PLB_MSSize[0:1]	I	1	The slave asserts this signal to indicate its native data width, which also indicates the maximum size of the data unit that can be transferred during each data beat on the bus. • `00`: 32 bits • `01`: 64 bits • `10`: 128 bits • `11`: Illegal
PLBPPCMTIMEOUT	PLB_MTimeout	I	0	The PLB arbiter drives this signal to each master. This signal is asserted in the 17th clock cycle after the assertion of PLB_PAValid, if no response is received from any slave. The crossbar then sends MIRQ back to the originating master.
PLBPPCMWRBTERM	PLB_MWrBTerm	I	0	The slave asserts this signal to indicate that the current burst write transfer in progress is to be terminated following the next write data beat.
PLBPPCMWRDACK	PLB_MWrDAck	I	0	The slave asserts this signal during a write transfer to indicate that the data currently on the PLB_wrDBus bus is no longer required by the slave (that is, the slave has either already latched the data or will latch the data at the end of the current clock cycle).

Table 4-2: **MPLB Interface Signals *(Cont'd)***

Port Name	Connects to PLB Signal	Dir	Default Value	Description
PLBPPCMWRPENDPRI[0:1]	PLB_wrPendPri[0:1]	I	x	Not used by the embedded processor block in Virtex-5 FPGAs. The PLB arbiter drives these signals, which are valid any time the PLB_wrPendReq signal is asserted. These signals indicate the highest priority of any active write request input from all masters attached to the PLB or a pipelined write transfer that has been acknowledged and is pending.
PLBPPCMWRPENDREQ	PLB_wrPendReq	I	0	Not used by the embedded processor block in Virtex-5 FPGAs. The PLB arbiter asserts this signal to indicate that a master has a write request pending on the PLB or a secondary write transfer has been acknowledged and is pending.
PPCMPLBABORT	M_abort	O		The MPLB interface never asserts this signal.
PPCMPLBABUS[0:31]	M_ABus[0:31]	O		32-bit starting address for the currently requested transfer. This bus is valid while M_request is active. For single-unit transfers, this bus indicates the address of the first enabled byte lane. For cache-line reads, this bus indicates the target word that should be returned first (optimally) when reading the line containing the address.
PPCMPLBBE[0:15]	M_BE[0:15]	O		For single-unit transfers, these signals act as byte enables to identify which bytes of the target being addressed are to be read from or written to. Each bit corresponds to a byte lane on the read or write data bus. For burst transfers, BE[0:3] indicates the number of data units of the requested size to be read or written, ranging from `0001` (2 units) to `1111` (16 units). It will never produce `0000` on BE[0:3] or a non-zero value on BE[4:15] for a burst transfer. This bus is not used during cache-line transfers (all zeros). This bus is valid while M_request is active. Due to command translation in the crossbar, the M_BE outputs do not necessarily reflect the PLB_BE signals received from the originating master.

Table 4-2: **MPLB Interface Signals *(Cont'd)***

Port Name	Connects to PLB Signal	Dir	Default Value	Description
PPCMPLBBUSLOCK	M_busLock	O		The current master can use the busLock signal to lock bus arbitration and force the PLB arbiter to continue to grant the bus to that master and ignore all other requests that are pending. This signal is propagated from the originating master.
PPCMPLBLOCKERR	M_lockErr	O		The master asserts this signal to indicate whether or not the slave must lock the Slave Error Address Register (SEAR) and the Slave Error Status Register (SESR). This signal is propagated from the originating master.
PPCMPLBMSIZE[0:1]	M_MSize[0:1]	O		These signals output a constant value (`10`) produced by the PPC440 wrapper to indicate that the MPLB interface is a 128-bit master.
PPCMPLBPRIORITY[0:1]	M_priority[0:1]	O		The master drives these signals to indicate to the PLB arbiter the priority of the master's request. These signals are valid while M_request is active.
PPCMPLBRDBURST	M_rdBurst	O		The master asserts this signal to indicate to the PLB arbiter that a burst read transfer is in progress. This signal is deasserted during the last data beat of the burst, as determined by the value of BE[0:3] at the time of the request.
PPCMPLBREQUEST	M_request	O		The master asserts this signal to request a data transfer across the PLB.
PPCMPLBRNW	M_RNW	O		This signal, which is driven by the master, is used to indicate whether the request is for a read (High) or a write (Low) transfer. This signal is valid while M_request is active.

Table 4-2: **MPLB Interface Signals *(Cont'd)***

Port Name	Connects to PLB Signal	Dir	Default Value	Description
PPCMPLBSIZE[0:3]	M_size[0:3]	O		This encoded value indicates the size and type of the requested transfer. Values produced by the MPLB are: • `0000`: Single-unit transfer of 1 to 16 bytes, as determined by BE[0:15] • `0001`: 4-word cache line • `0010`: 8-word cache line • `0011`: 16-word cache line (only if requested on SPLB by non Xilinx master) • `1100`: burst of quadword data units (number of data units to transfer indicated by BE[0:3]) This signal is valid while M_request is active. Due to command translation in the crossbar, the M_size outputs do not necessarily reflect the PLB_size signals received from the originating master.
PPCMPLBTATTRIBUTE[0]	M_TAttribute[0]	O		Write Through (W) Storage Attribute, as defined in the TLB when the processor is the master. This signal is valid while M_request is active. This signal is propagated from the originating master.
PPCMPLBTATTRIBUTE[1]	M_TAttribute[1]	O		Caching Inhibited (I) Storage Attribute, as defined in the TLB when the processor is the master. This signal is valid while M_request is active. This signal is propagated from the originating master.
PPCMPLBTATTRIBUTE[2]	M_TAttribute[2]	O		Memory Coherent (M) Storage Attribute, as defined in the TLB when the processor is the master. This signal is valid while M_request is active. This signal is propagated from the originating master.
PPCMPLBTATTRIBUTE[3]	M_TAttribute[3]	O		Guarded (G) Storage Attribute, as defined in the TLB when the processor is the master. This signal is valid while M_request is active. This signal is propagated from the originating master.
PPCMPLBTATTRIBUTE[4]	M_TAttribute[4]	O		U0 User-defined Storage Attribute, as defined in the TLB when the processor is the master. This signal is valid while M_request is active. This signal is propagated from the originating master.

Table 4-2: **MPLB Interface Signals *(Cont'd)***

Port Name	Connects to PLB Signal	Dir	Default Value	Description
PPCMPLBTATTRIBUTE[5]	M_TAttribute[5]	O		U1 User-defined Storage Attribute, as defined in the TLB when the processor is the master. This signal is optionally used to indicate transient cache region usage. This signal is valid while M_request is active. This signal is propagated from the originating master.
PPCMPLBTATTRIBUTE[6]	M_TAttribute[6]	O		U2 User-defined Storage Attribute, as defined in the TLB when the processor is the master. This signal is optionally used to indicate whether write cache misses allocate a cache line. This signal is valid while M_request is active. This signal is propagated from the originating master.
PPCMPLBTATTRIBUTE[7]	M_TAttribute[7]	O		U3 User-defined Storage Attribute, as defined in the TLB when the processor is the master. This signal is used by the crossbar as a Sync control signal, when asserted by any master targeting the MPLB, to block subsequent MPLB transfers until the target slave deasserts its MBusy signal for the MPLB master. The crossbar then propagates the slave's MBusy signal back to the originating master until it is deasserted by the slave. This allows masters to ensure that the transaction (such as a posted write) is actually completed by the slave before issuing subsequent transfers. This signal is valid while M_request is active. This signal is propagated from the originating master.
PPCMPLBTATTRIBUTE[8]	M_TAttribute[8]	O		User-defined attribute. Not used by the processor, but used in some systems to indicate an *Ordered Transfer* requirement. This signal is valid while M_request is active. This signal is propagated from the originating master.
PPCMPLBTATTRIBUTE[9:15]	M_TAttribute[9:15]	O		User-defined attributes (not used by the processor). These signals are valid while M_request is active. These signals are propagated from the originating master.
PPCMPLBTYPE[0:2]	M_type[0:2]	O		Not used by the processor. This bus is always driven to `000`, indicating a *memory* type transfer.
PPCMPLBUABUS[28:31]	M_UABus[28:31]	O		Upper 4 bits of the processor's 36-bit physical address, as specified by the Extended Real Page Number in the TLB. These 4 bits must always be set to `0000`.

Table 4-2: **MPLB Interface Signals *(Cont'd)***

Port Name	Connects to PLB Signal	Dir	Default Value	Description
PPCMPLBWRBURST	M_wrBurst	O		The master asserts this signal to indicate to the PLB arbiter that a burst write transfer is in progress. This signal is deasserted during the last data beat of the burst, as determined by the value of BE[0:3] at the time of the request.
PPCMPLBWRDBUS[0:127]	M_wrDBus[0:127]	O		128-bit bus used to transfer data from a master to a slave during a PLB write transfer.

MPLB Configuration

This section describes the two control registers in the embedded processor block in Virtex-5 FPGAs that control MPLB interface operation. Refer to the *PowerPC 440 Wrapper Data Sheet* [Ref 7] for a list of the parameters that control the default values of these registers.

The PLB Master configuration register (CFG_PLBM) is located at DCR address `0x54` (within the 256-word DCR address block allocated to the embedded processor block). Table 4-3 summarizes the bits in the CFG_PLBM register.

Table 4-3: **Bits in the CFG_PLBM Register**

Bits	Field	Default	Description
0	LOCK_SESR	1	SESR and SEAR are locked in DCR offsets `0x56`, `0x57`, and `0x58` (only allows updating when no prior error condition is detected).
1:22	Reserved	0	Reserved
23	Reserved	0	Must be set to 0.
24	XBAR_PRIORITY_ENA	1	• 0: Priority is disabled during crossbar arbitration • 1: Priority is enabled during crossbar arbitration
25	Reserved	0	Reserved
26	SL_ETERM_MODE	0	This bit must always be cleared to 0.
27	LOCKXFER	1	Lock Transfers. • 0: Disables lock transfers • 1: Enables lock transfers
28	RPIPE	1	Read Address Pipelining. • 0: Disables read address pipelining • 1: Enables read address pipelining Can be cleared directly or through bit 26.
29	WPIPE	1	Write Address Pipelining. • 0: Disables write address pipelining • 1: Enables write address pipelining Can be cleared directly or through bit 26. This bit is cleared if bit 30 is 0 to prevent posted write data.

Table 4-3: **Bits in the CFG_PLBM Register *(Cont'd)***

Bits	Field	Default	Description
30	WPOST	1	Write Posting. • 0: No write posting (early data acknowledgment) • 1: Enable write posting Bit 29 is cleared if this bit is 0. Only single transactions are supported if write posting is disabled. Interrupt status flag **`INT_CFG_ERR_M`** (DCR `0x20`, bit 17) is set if other types of transactions are received.
31	Reserved	1	Must be set to 1.

The PLB Master arbitration configuration register (ARB_XBC) is located at DCR address `0x23` (within the 256-word DCR address block allocated to the embedded processor block). Table 4-4 summarizes the bits in the ARB_XBC register.

This register configures crossbar arbitration priority and mode operations. All 32 bits have tie-off values. Arbitration priority values apply to fixed and round-robin arbitration only, with 4 corresponding to the highest priority and 0 corresponding to the lowest priority. Values between 5 and 7 are reserved and should not be used due to unpredictable behavior. The five device priority values should be mutually exclusive so that no two or more devices can have the same priority.

Table 4-4: **Bit Descriptions for the ARB_XBC Register**

Bits	Field	Default	Description
0:8	Reserved	`0`	Reserved
9:11	440ICUR	`100`	Instruction Read Priority
12	Reserved	`0`	Reserved
13:15	440DCUW	`011`	Data Write Priority
16	Reserved	`0`	Reserved
17:19	440DCUR	`010`	Data Read Priority
20	Reserved	`0`	Reserved
21:23	PLBS1	`000`	PLB Slave 1 Priority
24	Reserved	`0`	Reserved
25:27	PLBS0	`001`	PLB Slave 0 Priority
28	Reserved	`0`	Reserved
29	SYNCTATTR	`0`	Sync signal (TAttribute[7]) enable, if set; when reset, TAttribute[7] is ignored by the crossbar.
30:31	MODE	`00`	Arbitration Mode. • `00`: For Least Recently Used (LRU) • `01`: For round robin • `10`: For fixed priority • `11`: Reserved (should not be used due to unpredictable behavior)

SPLB Interfaces

The primary purpose of two crossbar SPLB interfaces is to allow PLB-based masters outside the embedded processor block in Virtex-5 FPGAs to share access to the main memory on the crossbar MCI. The crossbar is the primary means of establishing multiported access to the main memory in PowerPC 440 based systems. The SPLB interfaces also allow access to PLB-based memories and non-memory peripherals connected to the crossbar MPLB interface, which are also to be shared with the processor. However, users must exercise caution when connecting any type of bidirectional bridge-like device, such as a PCI bridge, between separate buses connected to MPLB and SPLB to avoid potential deadlock situations (see "PLB Interconnection Techniques," page 124). By default, XPS tools configure each SPLB address range to match only the crossbar MCI address space. Users must explicitly configure the embedded processor block to include the MPLB address spaces, if needed, in each SPLB address range.

The crossbar SPLB interface is a 128-bit wide slave on the PLB. The PLB to which it is connected is also populated by one or more masters, which can be of any mixture of data widths and functional capabilities. Due to the PLB interface limitations of the embedded processor block, a maximum of four masters can be connected to each SPLB interface. The SPLB can also be connected to the same PLB as the crossbar MPLB interface, as needed to achieve system connectivity requirements.

Transaction Types

The SPLB supports all transaction types allowed under the *PLB Architecture Specification* [Ref 4], which include:

- Single Unit

  The SPLB can accept single-unit transfers of unaligned data from 1 to 16 bytes. All single-unit transfers of any size that can be issued by all masters of any size and capability range are compatible with the SPLB interface.

- Bursts

  The SPLB can accept fixed length bursts of 2 to 16 data beats of aligned single words, doublewords, or quadwords. The crossbar can be configured to limit the maximum size, in quadwords, of all burst transfers propagated through the crossbar to be 16 (default), 8, 4, or 2. Any burst requests received by the SPLB that exceed this configured limit are internally broken into multiple bursts of up to the maximum size. All burst transfers that can be issued by all masters of any size are compatible with the SPLB interface.

- Cache Lines

  The SPLB can accept transfers of aligned 4- or 8-word (16- or 32-byte) cache lines. Cache-line reads are returned in the same order as received from the targeted slave, beginning either with the first word of the cache line or the quadword containing the target word, and always proceeds sequentially through the remainder of the cache line. All cache-line transfers that can be issued by all masters of any size are compatible with the SPLB interface.

  Four-word cache-line requests on the SPLB always appear as single transfers on the target master interface. The SPLB also automatically adjusts non-quadword aligned cache-line starting read addresses to quadword aligned addresses.

SPLB Interface Features

The SPLB can operate at an integer 1:N clock ratio with respect to the crossbar interconnect clock (CPMINTERCONNECTCLK), where 1 ≤N ≤16.

The SPLB never prematurely terminates a burst (by asserting rdBTerm/wrBTerm). If the targeted slave resides on the crossbar MPLB space and that slave, connected to the MPLB interface, prematurely terminates the burst, the MPLB interface continues to retry the residual command until that slave has completed the request.

For all transactions requested by narrower masters (32- or 64-bit native data width), the SPLB performs the required data steering to accept (write) or return (read) data in the byte lanes expected by the master.

The SPLB interface immediately acknowledges (asserts AddrAck) in the cycle following the sampling of the request signal from the master, when receiving requests that match its memory map, before the crossbar repeats the same request on the targeted crossbar master interface (MPLB or MCI).

By default, the SPLB interface immediately asserts WrDAck (in the same cycle as AddrAck) in response to write requests, stores data received from the master in its write data FIFO, and queues a posted write to the targeted crossbar master interface. Each SPLB interface can be configured to prohibit posted writes. If the SPLB is unable to acknowledge a request because of a command/data FIFO full condition, the SPLB asserts the rearbitrate signal.

The SPLB fully supports address pipelining, up to four read commands and four write commands. By default, the SPLB responds to the assertion of SAvalid from the PLB arbiter by asserting AddrAck. Each SPLB interface can be configured to disable address pipelining.

The SPLB supports overlapped read and write transfers. That is, a read request that is received (signaled by PAValid) while a prior write data transfer is in progress is immediately forwarded to the crossbar and can begin transferring data concurrently, and vice-versa.

SPLB Interface Signals

Table 4-5 summarizes the signals, in alphabetical order, of the SPLB 0 and SPLB 1 interfaces.

Table 4-5: **SPLB Interface Signals**

Port Name (n = [0, 1])	Connects to PLB Signal	Dir	Default Value	Description
CPMPPCSnPLBCLK	PLB_Clk	I	1	This clock synchronizes the SPLB interface to the connected PLB arbiter. The SPLB can operate at an integer 1:N clock ratio with respect to the crossbar interconnect clock (CPMINTERCONNECTCLK), where 1 ≤N ≤16.
PLBPPCSnBE[0:15]	PLB_BE[0:15]	I	x	For single-unit transfers, these signals act as byte enables to identify which bytes of the target being addressed are to be read from or written to. Each bit corresponds to a byte lane on the read or write data bus. For burst transfers, BE[0:3] indicates the number of data units of the requested size to be read or written, ranging from `0001` = 2 units to `1111` = 16 units. This bus is not used during cache-line transfers.
PLBPPCSnRNW	PLB_RNW	I	x	The master drives this signal to indicate whether the request is for a read (High) or a write (Low) transfer.
PLBPPCSnABORT	PLB_abort	I	0	The purpose of this signal is to indicate that the master no longer requires the data transfer it is currently requesting.
PLBPPCSnABUS[0:31]	PLB_ABus[0:31]	I	x	32-bit starting address for the currently requested transfer. For single-unit transfers, this bus indicates the address of the first enabled byte lane. For cache-line reads, this bus indicates the target word that should be returned first (optimally) when reading the line containing the address.
PLBPPCSnBUSLOCK	PLB_busLock	I	0	The current master can use the busLock signal to lock bus arbitration and force the PLB arbiter to continue to grant the bus to that master and ignore all other pending requests.
PLBPPCSnLOCKERR	PLB_lockErr	I	0	The master asserts this signal to indicate whether or not the slave must lock the Slave Error Address Register (SEAR) and the Slave Error Status Register (SESR).

Table 4-5: **SPLB Interface Signals *(Cont'd)***

Port Name (n = [0, 1])	Connects to PLB Signal	Dir	Default Value	Description
PLBPPCSnMASTERID[0:1]	PLB_masterID[0:1]	I	00	These signals from the arbiter indicate the master identification sequence number to the slave. The SPLB interface supports a maximum of four masters on the connected PLB.
PLBPPCSnMSIZE[0:1]	PLB_Msize[0:1]	I	01	The master drives these signals to indicate its native data width, which also indicates the maximum size of the data unit that can be transferred during each data beat on the bus. • `00`: 32 bits • `01`: 64 bits • `10`: 128 bits • `11`: Illegal
PLBPPCSnPAVALID	PLB_PAValid	I	0	The arbiter asserts this signal to indicate that there is a valid primary address and transfer qualifiers on the PLB outputs.
PLBPPCSnRDBURST	PLB_rdBurst	I	0	The master asserts this signal to indicate to the PLB arbiter that a burst read transfer is in progress. This signal is deasserted during the last data beat of the burst, as determined by the value of BE[0:3] at the time of the request.
PLBPPCSnRDPENDPRI[0:1]	PLB_rdPendPri[0:1]	I	x	Not used by the embedded processor block in Virtex-5 FPGAs. The arbiter uses these signals to indicate the highest priority of any active read request input from all masters attached to the PLB or a pipelined read transfer that has been acknowledged and is pending.
PLBPPCSnRDPENDREQ	PLB_rdPendReq	I	0	Not used by the embedded processor block in Virtex-5 FPGAs. The arbiter drives this signal to indicate that a master has a read request pending on the PLB or a secondary read transfer has been acknowledged and is pending
PLBPPCSnRDPRIM	PLB_rdPrim	I	0	The PLB arbiter asserts this signal to indicate that a secondary read request that has already been acknowledged by a slave can now be considered a primary read request.
PLBPPCSnREQPRI[0:1]	PLB_reqPri[0:1]	I	x	Not used by the embedded processor block in Virtex-5 FPGAs. The arbiter drives this signal to indicate the priority of the current request that the PLB arbiter has granted and is gating to the slaves.

Table 4-5: **SPLB Interface Signals *(Cont'd)***

Port Name (n = [0, 1])	Connects to PLB Signal	Dir	Default Value	Description
PLBPPCSnSAVALID	PLB_SAValid	I	0	The PLB arbiter asserts this signal to indicate to a PLB slave that there is a valid secondary or pipelined address and transfer qualifiers on the PLB outputs
PLBPPCSnSIZE[0:3]	PLB_size[0:3]	I	0	The encoded value of these signals indicates the size and type of the requested transfer. Supported values are: • `0000`: Single-unit transfer of 1 to 16 bytes, as determined by BE[0:15] • `0001`: 4-word cache line • `0010`: 8-word cache line • `0011`: 16-word cache line • `1010`: Burst of single-word data units • `1011`: Burst of doubleword data units • `1100`: Burst of quadword data units (number of data units to transfer for all bursts as indicated by BE[0:3]).
PLBPPCSnTATTRIBUTE[0]	M_TAttribute[0]	I	0	User-defined attribute. Used by the processor as the Write Through (W) Storage Attribute.
PLBPPCSnTATTRIBUTE[1]	M_TAttribute[1]	I	0	User-defined attribute. Used by the processor as the Caching Inhibited (I) Storage Attribute.
PLBPPCSnTATTRIBUTE[2]	M_TAttribute[2]	I	0	User-defined attribute. Used by the processor as the Memory Coherent (M) Storage Attribute.
PLBPPCSnTATTRIBUTE[3]	M_TAttribute[3]	I	0	User-defined attribute. Used by the processor as the Guarded (G) Storage Attribute.
PLBPPCSnTATTRIBUTE[4]	M_TAttribute[4]	I	0	User-defined attribute. Used by the processor to indicate the U0 Storage Attribute.
PLBPPCSnTATTRIBUTE[5]	M_TAttribute[5]	I	0	User-defined attribute. Used by the processor to indicate the U1 Storage Attribute.
PLBPPCSnTATTRIBUTE[6]	M_TAttribute[6]	I	0	User-defined attribute. Used by the processor to indicate the U2 Storage Attribute.

Table 4-5: **SPLB Interface Signals *(Cont'd)***

Port Name (n = [0, 1])	Connects to PLB Signal	Dir	Default Value	Description
PLBPPCSnTATTRIBUTE[7]	M_TAttribute[7]	I	0	Used by the crossbar as a Sync control signal when targeting the MPLB. Also used by the processor to indicate the U3 Storage Attribute. The Sync signal, if asserted at the time of the request, is used to block subsequent MPLB transfers until the target slave deasserts its MBusy signal back to the MPLB master. The crossbar then propagates the slave's MBusy signal back to the originating master on the SPLB interface until it is deasserted by the slave. This allows the originating masters to ensure that the transaction (such as a posted write) is completed by the slave before issuing subsequent transfers.
PLBPPCSnTATTRIBUTE[8]	M_TAttribute[8]	I	0	User-defined attribute. Not used by the embedded processor block in Virtex-5 FPGAs, but used in some systems to indicate an Ordered Transfer requirement. This signal is valid while M_request is active.
PLBPPCSnTATTRIBUTE[9:15]	M_TAttribute[9:15]	I	0	User-defined attributes (not used by the embedded processor block in Virtex-5 FPGAs).
PLBPPCSnTYPE[0:2]	PLB_type[0:2]	I	0	Not used by the embedded processor block in Virtex-5 FPGAs. These signals are always driven to `000`, indicating a memory type transfer.
PLBPPCSnUABUS[28:31]	PLB_UABus[28:31]	I	x	Upper 4 bits of the embedded processor block's 36-bit physical address. These 4 bits must always be set to `0000`.
PLBPPCSnWRBURST	PLB_wrBurst	I	0	The master drives this signal to indicate to the PLB arbiter that a burst write transfer is in progress. This signal is deasserted during the last data beat of the burst, as determined by the value of BE[0:3] at the time of the request.
PLBPPCSnWRDBUS[0:127]	PLB_wrDBus[0:127]	I	x	128-bit bus used to transfer data between a master and a slave during a PLB write transfer.

Table 4-5: **SPLB Interface Signals *(Cont'd)***

Port Name (n = [0, 1])	Connects to PLB Signal	Dir	Default Value	Description
PLBPPCSnWRPENDPRI[0:1]	PLB_wrPendPri[0:1]	I	x	Not used by the embedded processor block in Virtex-5 FPGAs. The arbiter drives these signals to indicate the highest priority of any active write request input from all masters attached to the PLB or a pipelined write transfer that has been acknowledged and is pending.
PLBPPCSnWRPENDREQ	PLB_wrPendReq	I	0	Not used by the embedded processor block in Virtex-5 FPGAs. The arbiter drives this signal to indicate that a master has a write request pending on the PLB or a secondary write transfer has been acknowledged and is pending.
PLBPPCSnWRPRIM	PLB_wrPrim	I	0	The PLB arbiter asserts this signal to indicate that a secondary or pipelined write request might be considered a primary write request in the clock cycle that follows.
PPCSnPLBADDRACK	Sl_addrAck	O		This signal is asserted to indicate that the slave has acknowledged the address and will latch the address and all of the transfer qualifiers at the end of the current clock cycle.
PPCSnPLBMBUSY[0:3]	Sl_MBusy[0:3]	O		These signals indicate to each connected master (according to the value of PLB_masterID[0:1]) that the SPLB interface is either busy performing a read or a write transfer, or has a read or write transfer pending for that master. The SPLB interface supports a maximum of four masters on the connected PLB. When the current transfer targets the MPLB interface, MBusy is propagated back from the targeted slave until it is deasserted by the slave (if the Sync signal was asserted at the time of the request) or until the current transfer completes on the MPLB (if Sync was not asserted, see M_TAttribute[7]).

Table 4-5: **SPLB Interface Signals *(Cont'd)***

Port Name (n = [0, 1])	Connects to PLB Signal	Dir	Default Value	Description
PPCSnPLBMIRQ[0:3]	Sl_MIRQ[0:3]	O		These signals indicate to each connected master (according to the value of PLB_masterID[0:1]) that the SPLB interface has encountered an event which it has deemed important to the master. MIRQ is generally used to indicate an error condition not associated with a read or write transfer currently in progress, for example, a write error or time-out during a posted write to the targeted slave. MIRQ is propagated back from the targeted slave to the originating master, if known. Otherwise it is broadcast to all PLB masters connected to the crossbar (including the processor). The SPLB interface supports a maximum of four masters on the connected PLB.
PPCSnPLBMRDERR[0:3]	Sl_MRdErr[0:3]	O		These signals indicate to each connected master (according to the value of PLB_masterID[0:1]) that the SPLB interface has encountered an error during a read transfer that was initiated by this master, typically a transfer that is still in progress. When the current transfer targets the MPLB interface, MRdErr is propagated back from the targeted slave. The SPLB interface supports a maximum of four masters on the connected PLB.
PPCSnPLBMWRERR[0:3]	Sl_MWrErr[0:3]	O		These signals indicate to each connected master (according to the value of PLB_masterID[0:1]) that the SPLB interface has encountered an error during a write transfer that was initiated by this master, typically a transfer that is still in progress. When the current transfer targets the MPLB interface, MWrErr is propagated back from the targeted slave if the transfer is still in progress (not posted). The SPLB interface supports a maximum of four masters on the connected PLB.
PPCSnPLBRDBTERM	Sl_rdBTerm	O		This signal indicates to a master that the current burst read transfer in progress is to be terminated following the next read data beat. This signal is normally asserted only during the second-to-last beat of a fixed-length read burst, as determined by the value of BE[0:3] at the time of the request.

Table 4-5: **SPLB Interface Signals *(Cont'd)***

Port Name (n = [0, 1])	Connects to PLB Signal	Dir	Default Value	Description
PPCSnPLBRDCOMP	Sl_rdComp	O		This signal indicates to the PLB arbiter that the read transfer is either complete or will be complete by the end of the next clock cycle.
PPCSnPLBRDDACK	Sl_rdDAck	O		This signal indicates that the data on the Sl_rdDBus bus is valid and must be latched at the end of the current clock cycle.
PPCSnPLBRDDBUS[0:127]	Sl_rdDBus[0:127]	O		128-bit data bus used to transfer data from a slave and to a master during a PLB read transfer.
PPCSnPLBRDWDADDR[0:3]	Sl_rdWdAddr[0:3]	O		These signals indicate the relative word address within the cache line of the first word of the data unit currently being transferred as part of a requested cache-line read transfer. Cache-line reads are returned in the same order as received from the targeted slave, beginning either with the first word of the cache line or the data unit (word, doubleword, or quadword aligned, depending on the targeted slave's data width) containing the target word, and always proceeds sequentially through the remainder of the cache line, incrementing according to the size of the data units being transferred.
PPCSnPLBREARBITRATE	Sl_rearbitrate	O		This signal is asserted to indicate that the slave cannot perform the currently requested transfer and requires the PLB arbiter to re-arbitrate the bus.
PPCSnPLBSSIZE[0:1]	Sl_SSize[0:1]	O		These signals are driven to `10` to indicate that the SPLB interface is a 128-bit slave.
PPCSnPLBWAIT	Sl_wait	O		The SPLB interface never asserts this signal. This signal is used by other PLB slaves to indicate to the arbiter that the slave has recognized the PLB address as a valid address, but cannot latch the address and all the transfer qualifiers at the end of the current clock cycle.
PPCSnPLBWRBTERM	Sl_wrBTerm	O		This signal indicates to a master that the current burst write transfer in progress is to be terminated following the next write data beat. This signal is normally asserted only during the second-to-last beat of a fixed-length write burst, as determined by the value of BE[0:3] at the time of the request.

Table 4-5: **SPLB Interface Signals *(Cont'd)***

Port Name (n = [0, 1])	Connects to PLB Signal	Dir	Default Value	Description
PPCSnPLBWRCOMP	Sl_wrComp	O		This signal indicates to the arbiter the end of the current write transfer.
PPCSnPLBWRDACK	Sl_wrDAck	O		This signal indicates that the data currently on the PLB_wrDBus bus is no longer required by the slave (write data is being latched at the end of the current clock cycle).

SPLB Configuration

One control register in the embedded processor block in Virtex-5 FPGAs controls the operation of each of the SPLB interfaces. The PLB Slave configuration register for SPLB 0 (CFG_PLBS0) is located at DCR address `0x34`. The configuration register for SPLB 1 (CFG_PLBS1) is located at DCR address `0x44` (within the 256-word DCR address block allocated to the embedded processor block), with default values as shown in Table 4-6. Each of these registers also contain the master enable bits for two of the four DMA controllers in the embedded processor block, which do not affect SPLB interface operation. Refer to the *PowerPC 440 Wrapper Data Sheet* [Ref 7] for a list of the parameters that control the default values of these registers.

Table 4-6: **Bit Descriptions for the CFG_PLBS0/1 Registers**

Bits	Field	Default	Description
0	LOCK_SESR	`1`	Lock SESR and SEAR if set (only allows updating when no prior error condition has been detected). For CFG_PLBS0, locks DCR `0x36`, `0x37`, and `0x38`. For CFG_PLBS1, locks DCR `0x46`, `0x47`, and `0x48`.
1	Reserved	`0`	Reserved
2	DMA1_EN (PLBS0) DMA3_EN (PLBS1)	`0`	DMA controller master enable 1. For CFG_PLBS0: Enable DMA #1 controller. For CFG_PLBS1: Enable DMA #3 controller.
3	DMA0_EN (PLBS0) DMA2_EN (PLBS1)	`0`	DMA controller master enable 0. For CFG_PLBS0: Enable DMA #0 controller. For CFG_PLBS1: Enable DMA #2 controller.
4:5	DMA0_PRI (PLBS0) DMA2_PRI (PLBS1)	`00`	DMA priority value 0. For CFG_PLBS0: Priority of DMA #0 controller. For CFG_PLBS1: Priority of DMA #2 controller.
6:7	DMA1_PRI (PLBS0) DMA3_PRI (PLBS1)	`00`	DMA priority value 1. For CFG_PLBS0: Priority of DMA #1 controller. For CFG_PLBS1: Priority of DMA #3 controller.
8	Reserved	`0`	Reserved

Table 4-6: **Bit Descriptions for the CFG_PLBS0/1 Registers *(Cont'd)***

Bits	Field	Default	Description
9:11	THRMIB	`011`	Maximum burst threshold when reading from the memory interface. • `000`: Threshold of 1, a burst is translated into single transfers. • `001`: Threshold of 2, a burst is translated into bursts of maximum length 2, if applicable. • `010`: Threshold of 4, a burst is translated into bursts of maximum length 4, if applicable. • `011`: Threshold of 8, a burst is translated into bursts of maximum length 8, if applicable. • `100`: Threshold of 16, a burst is translated into bursts of maximum length 16, if applicable. • `101` to `111`: Reserved values that can cause unpredictable behaviors.
12	Reserved	`0`	Reserved
13:15	THRPLBM	`011`	Maximum burst threshold when reading from MPLB. • `000`: Threshold of 1, a burst is translated into single transfers. • `001`: Threshold of 2, a burst is translated into bursts of maximum length 2, if applicable. • `010`: Threshold of 4, a burst is translated into bursts of maximum length 4, if applicable. • `011`: Threshold of 8, a burst is translated into bursts of maximum length 8, if applicable. • `100`: Threshold of 16, a burst is translated into bursts of maximum length 16, if applicable. • `101` to `111`: Reserved values that can cause unpredictable behaviors.
16	Reserved	`0`	Reserved
17:19	THWMIB	`011`	Maximum burst threshold when writing to the memory interface. • `000`: Threshold of 1, a burst is translated into single transfers. • `001`: Threshold of 2, a burst is translated into bursts of maximum length 2, if applicable. • `010`: Threshold of 4, a burst is translated into bursts of maximum length 4, if applicable. • `011`: Threshold of 8, a burst is translated into bursts of maximum length 8, if applicable. • `100`: Threshold of 16, a burst is translated into bursts of maximum length 16, if applicable. • `101` to `111`: Reserved values that can cause unpredictable behaviors.
20	Reserved	`0`	Reserved

Table 4-6: **Bit Descriptions for the CFG_PLBS0/1 Registers *(Cont'd)***

Bits	Field	Default	Description
21:23	THWPLBM	`011`	Maximum burst threshold when writing to MPLB. • `000`: Threshold of 1, a burst is translated into single transfers. • `001`: Threshold of 2, a burst is translated into bursts of maximum length 2, if applicable. • `010`: Threshold of 4, a burst is translated into bursts of maximum length 4, if applicable. • `011`: Threshold of 8, a burst is translated into bursts of maximum length 8, if applicable. • `100`: Threshold of 16, a burst is translated into bursts of maximum length 16, if applicable. • `101` to `111`: Reserved values that can cause unpredictable behaviors.
24	Reserved	`0`	Reserved
25	LOCKXFER	`1`	Lock Transfers. • 0: Disables lock transfers • 1: Enables lock transfers
26	RPIPE	`1`	Read Address Pipelining. • 0: Disables read address pipelining • 1: Enables read address pipelining
27	WPIPE	`0`	Write Address Pipelining. • 0: Disables write address pipelining • 1: Enables write address pipelining Can be cleared directly or through bit 28.
28	WPOST	`1`	Write Posting. • 0: No write posting (early data acknowledgment) • 1: Write posting enabled Bit 27 is cleared if this bit is 0. Only single transactions are supported if write posting is disabled. The corresponding interrupt status flag (in the Interrupt status register at DCR `0x20`) is set if other types of transactions are received: INT_CFG_ERR_S0 (bit 3) for SPLB 0, or INT_CFG_ERR_S1 (bit 11) for SPLB 1.
29	Reserved	`1`	Must be set to 1.
30	AERR_LOG	`0`	Log ABUS address mismatch error, if set, in the PLB slave miscellaneous status register (bit 2 of DCR `0x39` or `0x49`)
31	CMD_CHK_DBL	`0`	Disable command (size) check, if set, in the PLB slave miscellaneous status register (bits 0 and 1 of DCR `0x39` or `0x49`)

In addition to the control register, each SPLB interface has a set of address template registers (TMPL*_PLBS*_MAP at DCR addresses `0x40` - `0x4F` and `0x50` - `0x5F`) that determine to which addresses the SPLB is to respond. The SPLB interface acknowledges requests based on the settings of these registers rather than waiting for any response from the targeted slave back through the crossbar. These template registers are normally set automatically by the EDK software, based on the addresses assigned to slaves connected to the crossbar. By default, EDK sets the template registers so that only the address range of the memory connected to the crossbar MCI (if any) is accessible via the SPLB, to avoid potential deadlock situations (see "PLB Interconnection Techniques," page 124). Users can override this setting to include addresses of slaves connected to the MPLB interface.

Command Translation

All data traffic through the crossbar is packed into 128-bit quadword-aligned units. However, each of the SPLB and MPLB interfaces may connect to masters and slaves that have various native widths of 32, 64, or 128 bits. For each data transfer (command) requested by one of the masters connected to the crossbar (processor, SPLB interface, DMA controller), command translation may be required to handle any combination of the following conditions:

1. A master connected to an SPLB interface has a native data width smaller than 128 bits and requires byte-lane steering or extra cycles to pack data into 128-bit units.
2. The slave's native data width (connected to MPLB or MCI) is smaller than 128 bits and requires byte-lane steering or mirroring and/or extra cycles to unpack from 128 bits.
3. The length of a burst has to be adjusted for mismatched bus widths.
4. The maximum length of a burst has been exceeded and requires breaking into multiple commands.
5. Misaligned burst transfer beginning or ending off of a quadword boundary requires a single-unit transfer (up to three words) to be generated at the beginning and/or end of the transfer.

Each of the crossbar master interfaces (MPLB and MCI) maintains a command queue that can store several requests from the various crossbar sources (processor, SPLB, DMA) awaiting access to the connected PLB or memory controller. Commands in each queue are issued only in the order which they are queued.

The re-arbitrate signal asserted by a PLB slave only allows the arbiter of that PLB to grant access to a different master requesting the bus. If the crossbar MPLB receives a re-arbitrate signal while requesting the bus, it backs off temporarily, as required, but always resumes with the request of the same command at the head of its command queue.

The crossbar's command queues and internal data FIFOs allow it to adapt mismatched transfer rates among its various slave and master interfaces. Transfer rate mismatch can occur due to any combination of the following:

- Differing clock frequencies among the crossbar, the CPU and each of the SPLB, DMA, MPLB, and MCI interfaces.
- Multiple sources (CPU, SPLB, DMA) arbitrating for the same master interface (MPLB or MCI).
- Packing and unpacking data to interface to devices with native data widths less than 128 bits connected to the MPLB or SPLB.
- Additional arbitration latency introduced when long bursts are broken into multiple shorter bursts.
- Inherent latency of gaining access to the PLB connected to the MPLB interface, and the response times of the slaves on that bus or of the memory controller connected via the MCI.

Crossbar Timing

Crossbar Latency

The SPLB performs a store-and-forward operation for write commands, accordingly the latency depends on the type of transfer and the number of beats involved in the transfer. The minimum latency for a write or a read command presented to the SPLB interface to traverse the embedded block is six cycles. For read transactions, the embedded processor block adds another two or three cycles of latency to the return data path. Figure 4-1 shows a simplified timing model of the latencies for different types of transactions between SPLB and MPLB interfaces. Table 4-7 defines the values of n and m used in Figure 4-1.

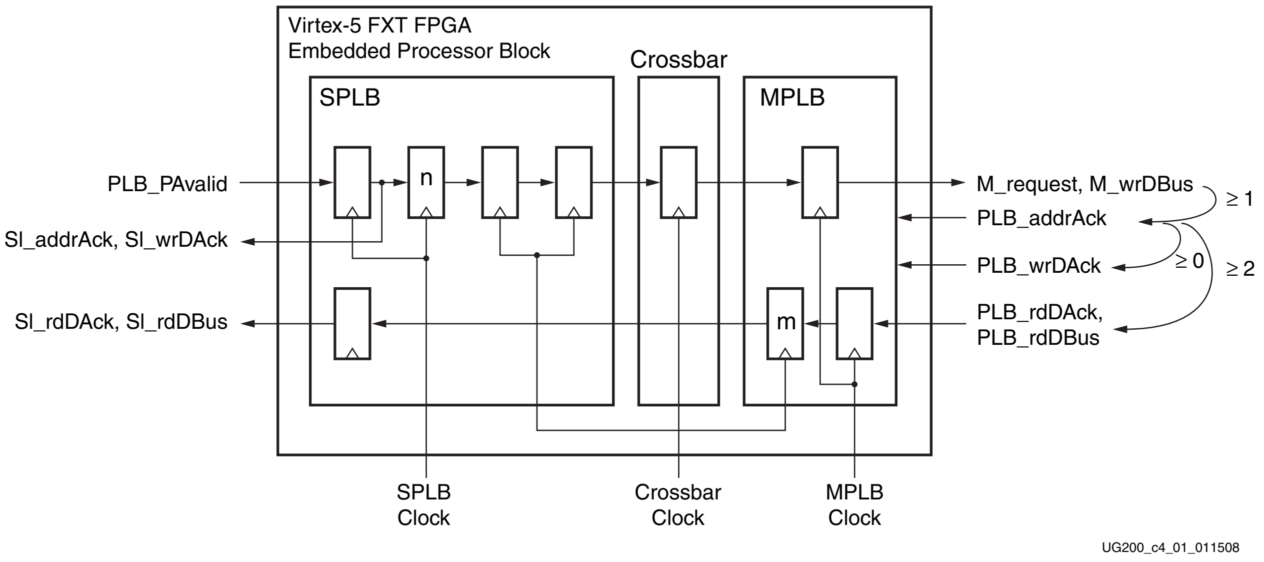


Figure 4-1: **Simplified Crossbar Timing Model**

Table 4-7: **Values of Symbols in Figure 4-1**

Transfer Type	n	m
Write Single-unit	1	
Write Line	Line Length – 1	
Write Burst	Burst Length	
Read Single-unit	1	0
Read Line	1	0
Read Burst	2	1

Notes:

1. Line Length and Burst Length refer to the actual number of data beats required to transfer that data across the SPLB interface (before command translation by the crossbar), and therefore increases if the natural data width of the originating master is reduced to 64 or 32 bits.

When a request is received on the SPLB0/SPLB1 interface from the connected PLB arbiter, the interface responds by asserting AddrAck in the next cycle (otherwise it asserts the re-arbitrate signal if its FIFO is full). Provided there is no arbitration contention in the crossbar for the targeted master interface (MPLB or MCI), and provided there are no prior unacknowledged requests queued at either the SPLB or the master interface, and provided the SPLB, crossbar, and master interface are running at the same clock frequency (1:1:1 clock ratio), a request is propagated from the SPLB onto the master interface after an additional five or more cycles, as indicated in Table 4-7.

The actual completion of each transfer on the MPLB side depends on the availability and response times of the connected PLB arbiter and target slave. Typically, the PLB arbiter introduces a minimum of one cycle latency between M_request and the assertion of PAvalid. PLB_addrAck can therefore be returned in one or more cycles, as is required. (The MPLB interface should not be used in a manner that would allow PLB_addrAck to be asserted during the same cycle as M_request.) For write transfers, PLB_wrDAck can be asserted at the same time as PLB_addrAck, or later. According to PLB v4.6 protocol, PLB_RdDAck must not be asserted prior to the second cycle following PLB_addrAck.

Transaction Waveforms

Figure 4-2 through Figure 4-12 show the waveforms for various transactions involving the SPLB and MPLB interfaces. While all are shown as read or write transfers between a master on the SPLB and a slave on the MPLB, the waveforms are representative of transfers originating at any of the crossbar slave interfaces (SPLB, processor PLB, DMA) or targeting either of the crossbar master interfaces (MPLB or MCI). In the following waveform figures, a trace in the center of a bus signal signifies values that are undetermined (in the context of the given transaction), not a high-impedance state.

Figure 4-2 shows a typical burst write transfer. Because the SPLB interface controller buffers all data before forwarding the command through crossbar, the latency varies with the length of the burst (number of data beats on the SPLB interface). In this case, a four unit burst results in nine cycles of latency between the SPLB and MPLB interfaces. Best-case waveforms are shown for response times of the connected PLB arbiter and target slave.

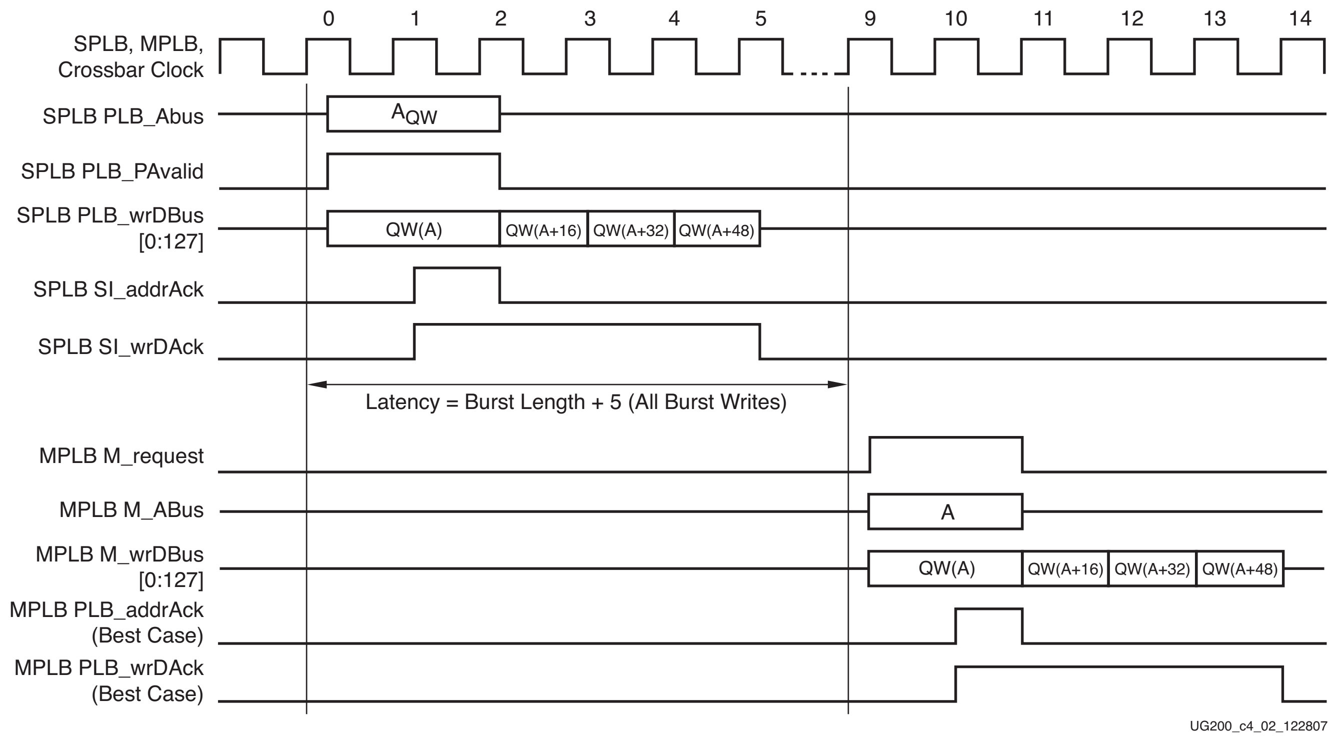


Figure 4-2: **SPLB to MPLB 4-Quadword Burst Write Transaction**

Figure 4-3 shows a typical burst read transfer. The latency to propagate burst read commands to the MPLB is constant, seven cycles. Once the target slave responds, read data traverses back to the SPLB interface in three cycles for all burst transfers.

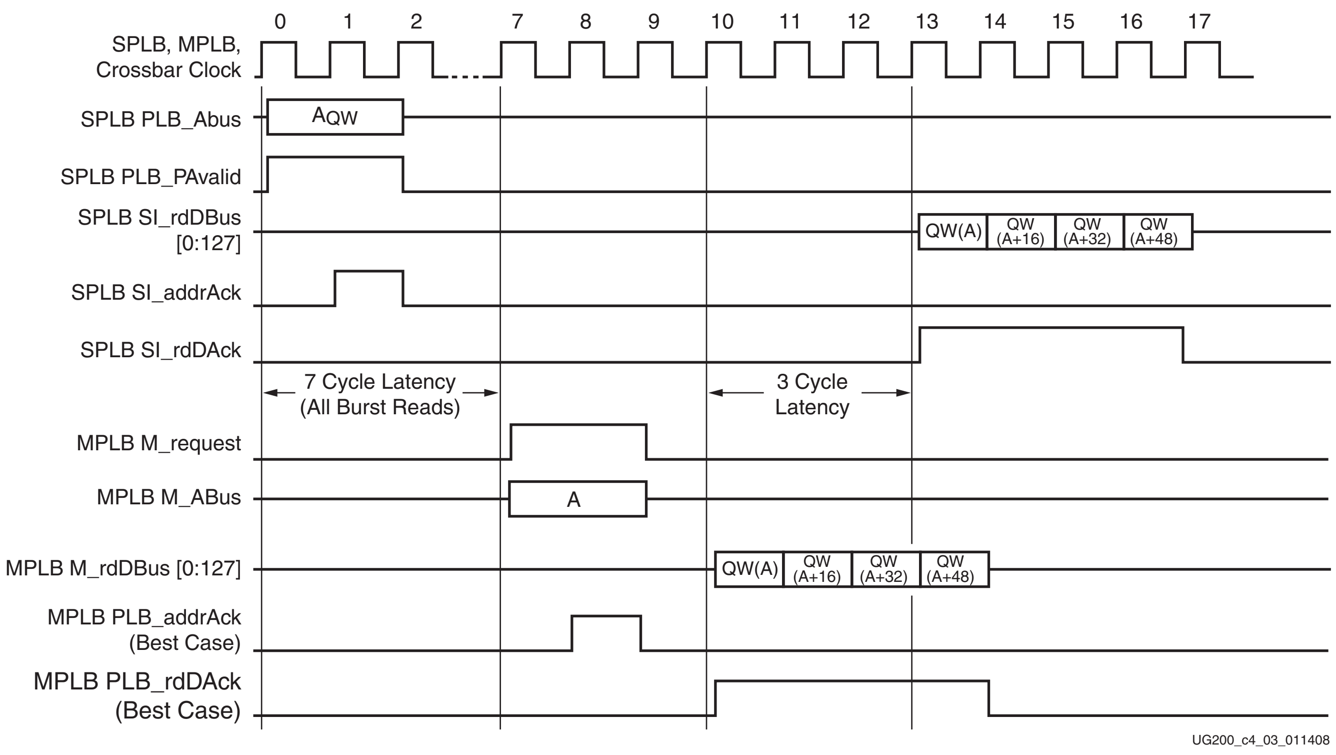


Figure 4-3: **SPLB to MPLB 4-Quadword Burst Read Transaction**

Figure 4-4 shows how the number of data beats in a write burst transfer are multiplied when a slave smaller than 128 bits responds on the MPLB interface. The MPLB unpacks the two quadwords into 8 single words and steers them onto the lower byte lanes of the write data bus for the 32-bit slave. The write from the SPLB is posted, so the SPLB completes its transaction as soon as the two quadwords are transferred, unaffected by the size of the slave responding on the MPLB.

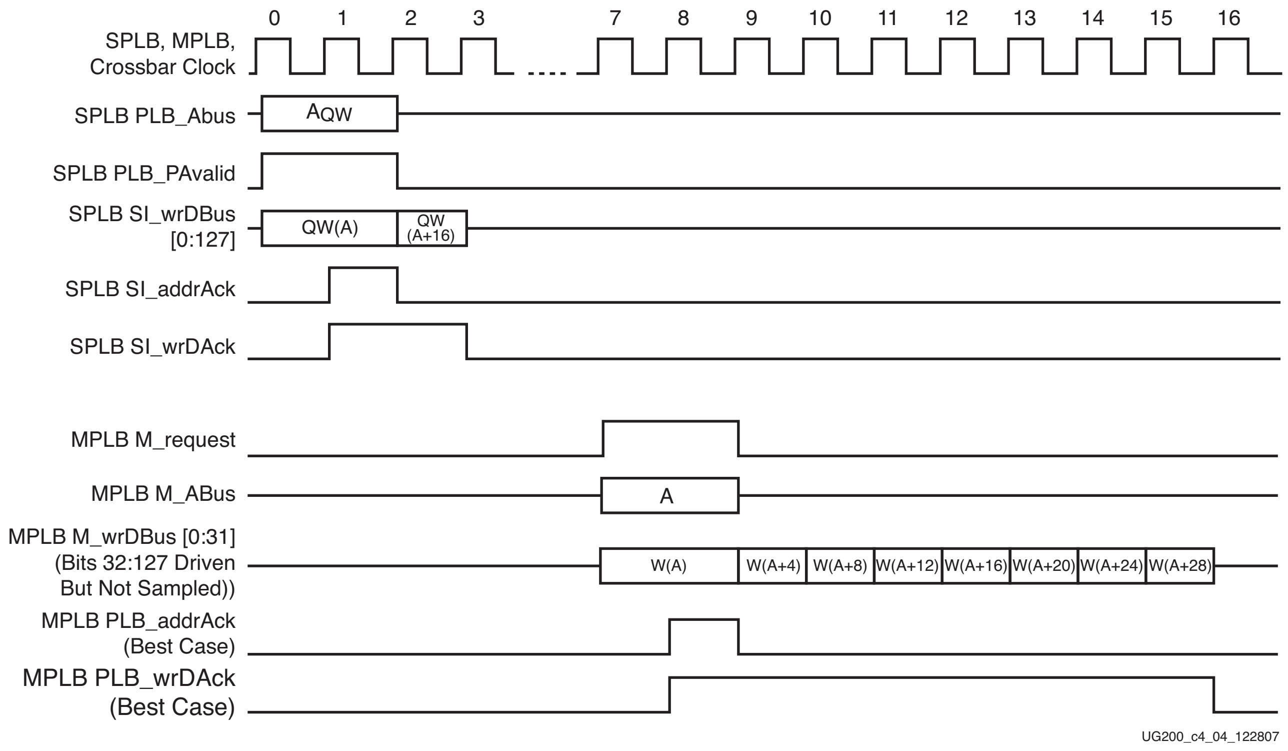


Figure 4-4: **SPLB to MPLB 2-Quadword Burst Write by 128-Bit Master to 32-Bit Slave**

Figure 4-5 shows how a burst read command from a 64-bit master is converted by the SPLB interface into a 128-bit burst (half the number of data beats) when propagating through the crossbar. The two quadwords returned from the MPLB are then unpacked by the SPLB interface and steered onto the lower byte lanes of the read data bus for the 64-bit master.

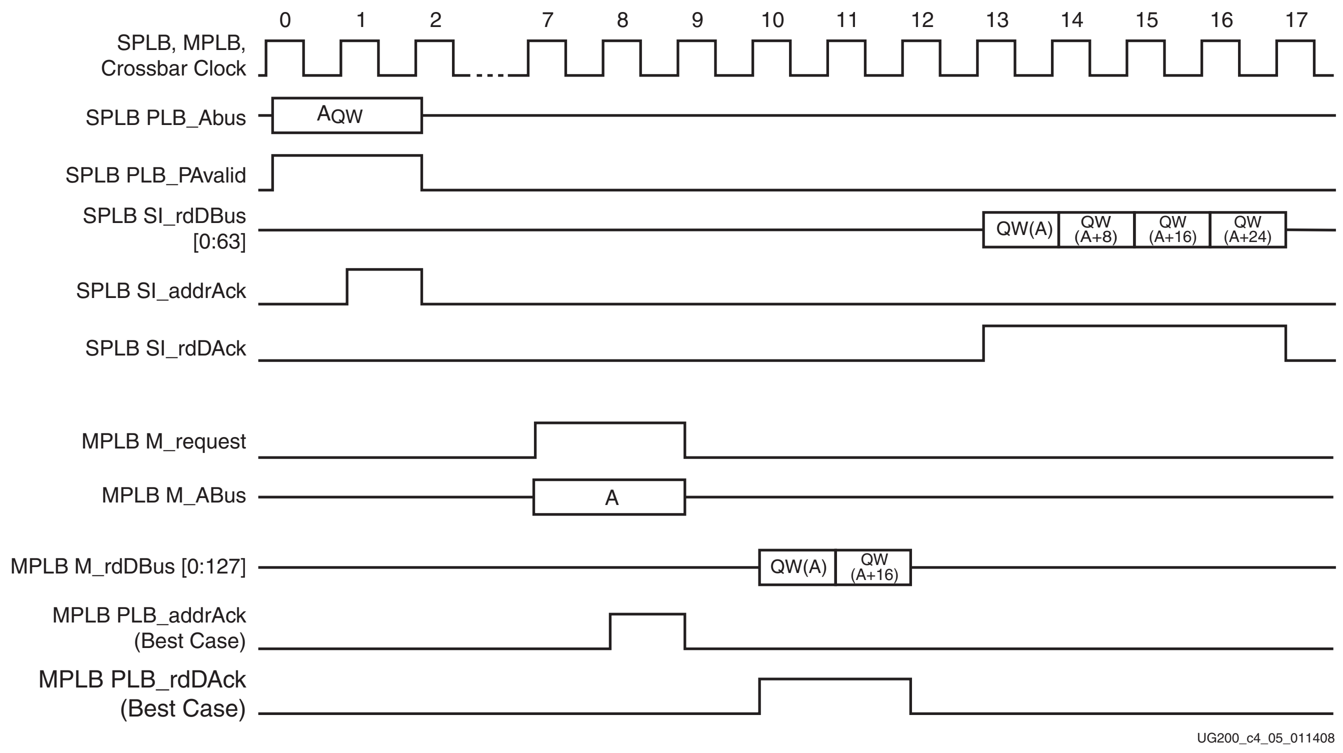


Figure 4-5: **SPLB to MPLB 4-Doubleword Burst Read by 64-Bit Master from 128-Bit Slave (Aligned)**

Figure 4-6 shows the conversion of a single-unit write command by the 128-bit MPLB to a 32-bit slave. Single-unit write and read commands propagate to the SPLB in 6 cycles (fixed). During the first write request by the MPLB, the entire quadword is presented on the write data bus, since the MPLB interface does not yet know the size of the slave that will respond. The 32-bit slave actually reads bytes 5 through 7, which are part of the single word that the ABus is pointing to. At that time, the MPLB interface knows that it must request additional write commands to transfer the remaining two single words to the 32-bit slave. The ABus is incremented accordingly, but the MPLB still populates the write data bus and byte enables so that all data is still presented at the proper locations for any size slave.

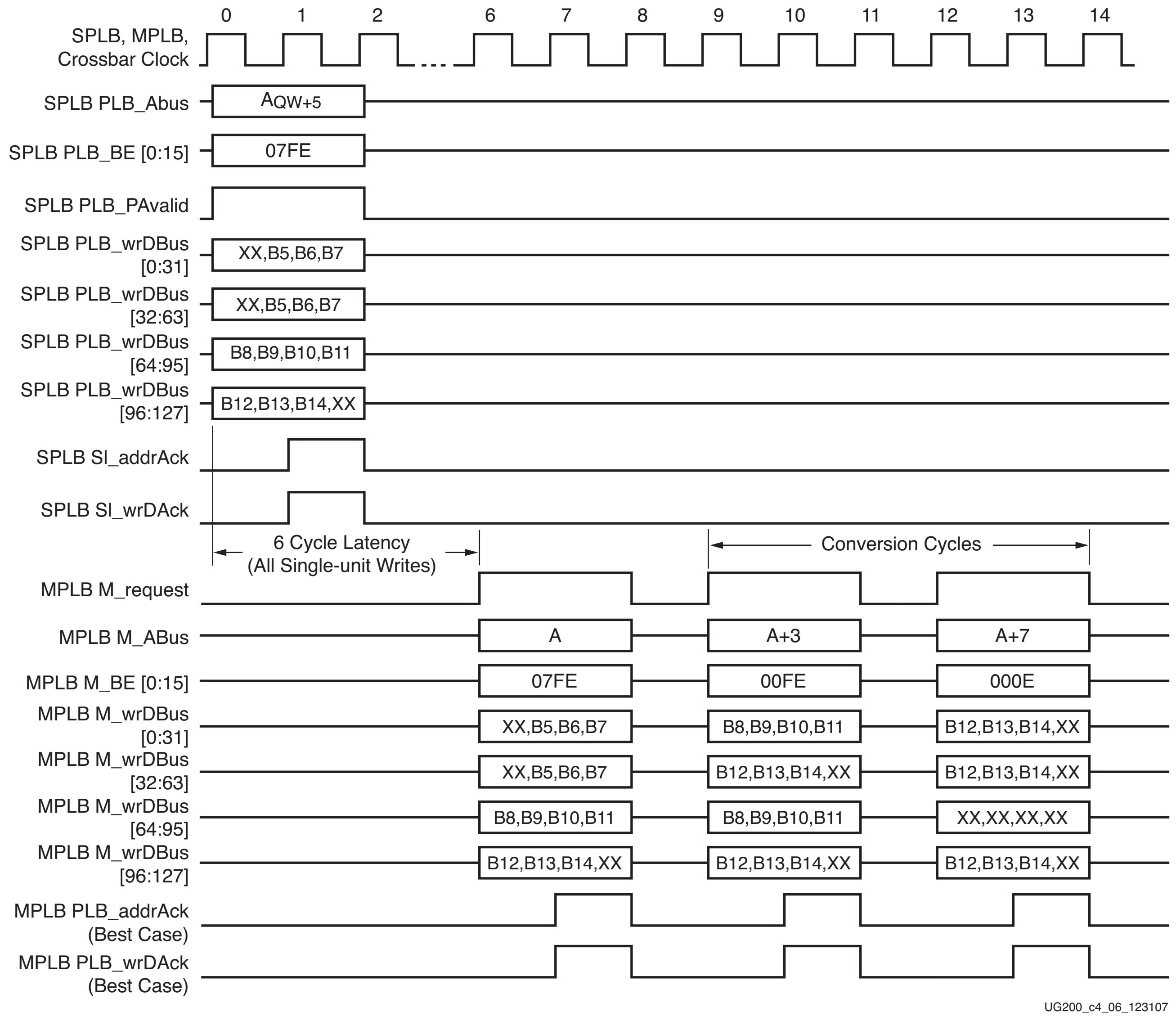


Figure 4-6: **SPLB to MPLB Single-Unit Write of Bytes 5-14 by 128-Bit Master to 32-Bit Slave**

Figure 4-7 shows the conversion of a single-unit read command by the 128-bit MPLB to a 64-bit slave. Even though the original command received from the SPLB requested a doubleword, the address is not aligned on a doubleword boundary. Consequently, the MPLB needs to split the command into two single-unit reads. Before propagating the two words received from the slave back across the crossbar, the MPLB packs them into a single quadword unit, as is expected by the SPLB.

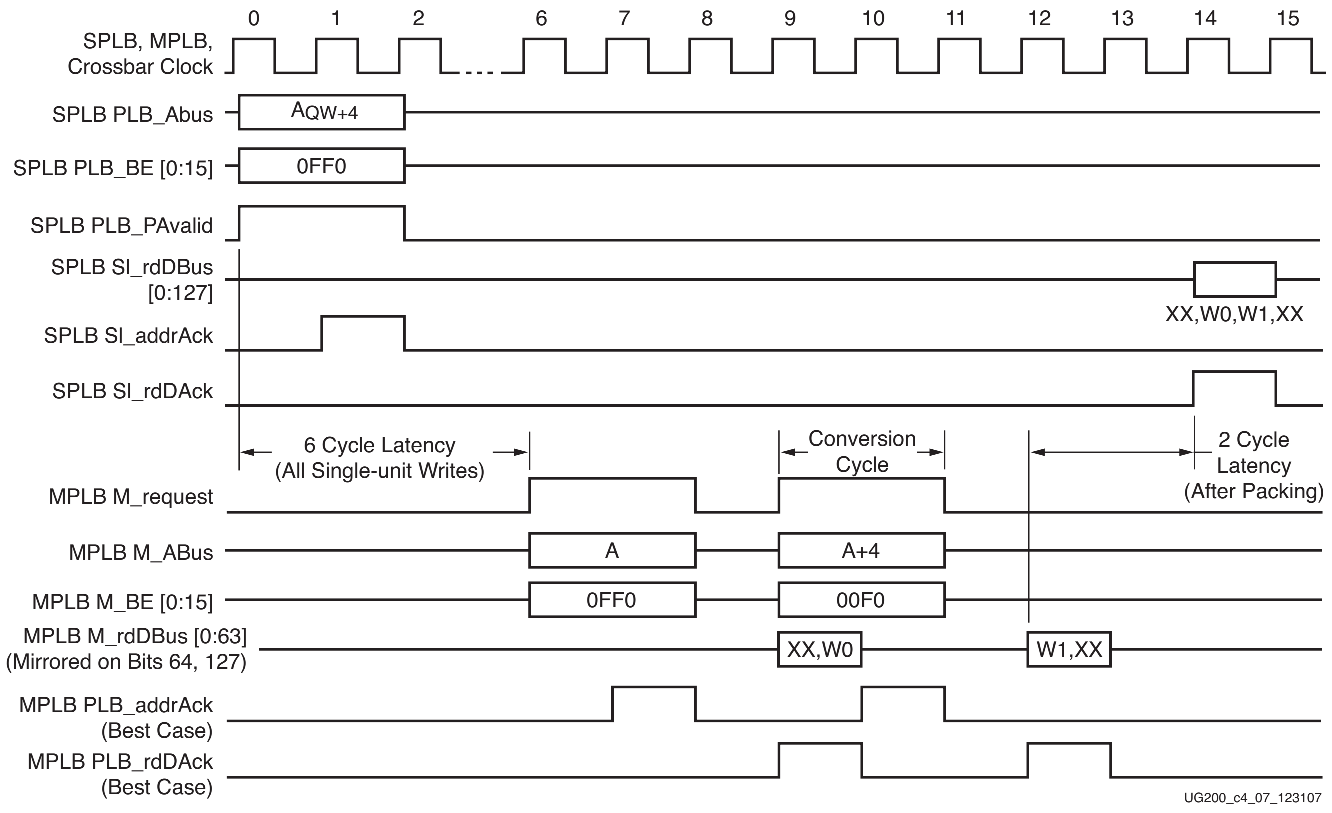


Figure 4-7: **SPLB to MPLB Single-Unit Read of 2 Unaligned Words by 128-Bit Master from 64-Bit Slave**

Figure 4-8 shows a burst of six doubleword data units, where the beginning and ending of the burst do not align on quadword boundaries. In the SPLB interface, the burst is translated into a sequence of quadword-aligned commands. The unaligned start of the burst is translated into a single-unit transfer of eight bytes (doubleword 0), followed by a burst transfer of two aligned quadwords (doublewords 1 through 4). The sequence is then completed using another single-unit transfer of 8 bytes (doubleword 5). Upon receiving the three packets of data, the SPLB interface unpacks the data and presents a burst of six doublewords on the read data bus, as originally requested.

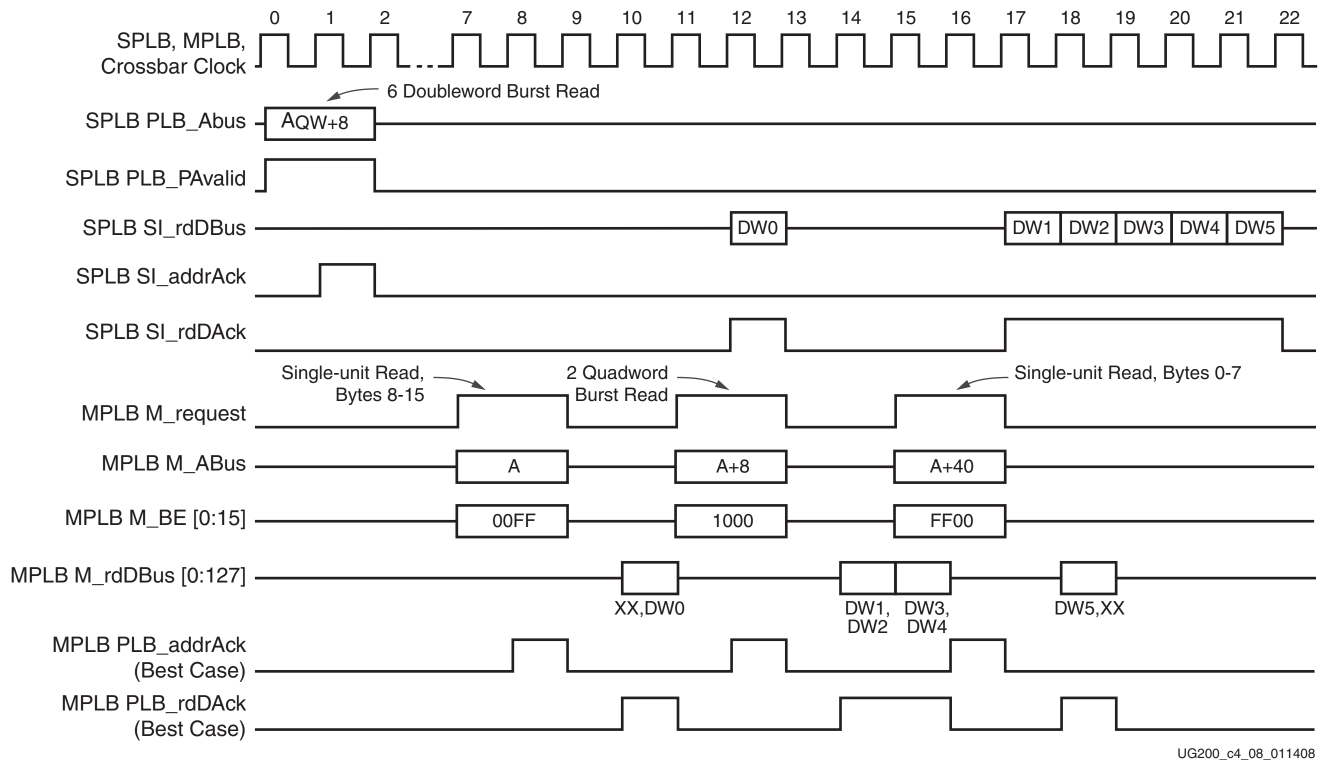


Figure 4-8: **SPLB to MPLB 6-Doubleword Unaligned Burst Read from 128-Bit Slave (Requested by a 64-Bit Master)**

Figure 4-9 shows back-to-back write burst requests. The second request (address B) is made at the same time as the first data transfer completes. Consequently it is presented as a primary request (PAvalid) by the arbiter. This timing is optimal in that it allows the write data bus to continually transfer data. On the MPLB interface, the resulting sequence of write requests typically leaves a 1-cycle bubble on the write data bus between each successive command (assuming best-case arbiter/slave response).

If the second request (address B) is instead issued on the SPLB before the completion of the first transfer, it would result in the pipelining of a secondary request. As the first data transfer completes, the arbiter asserts PLB_wrPrim to inform the SPLB interface to begin buffering the write data for the second transfer. However, the command latency and waveforms presented on the MPLB interface typically remain the same as shown in Figure 4-9.

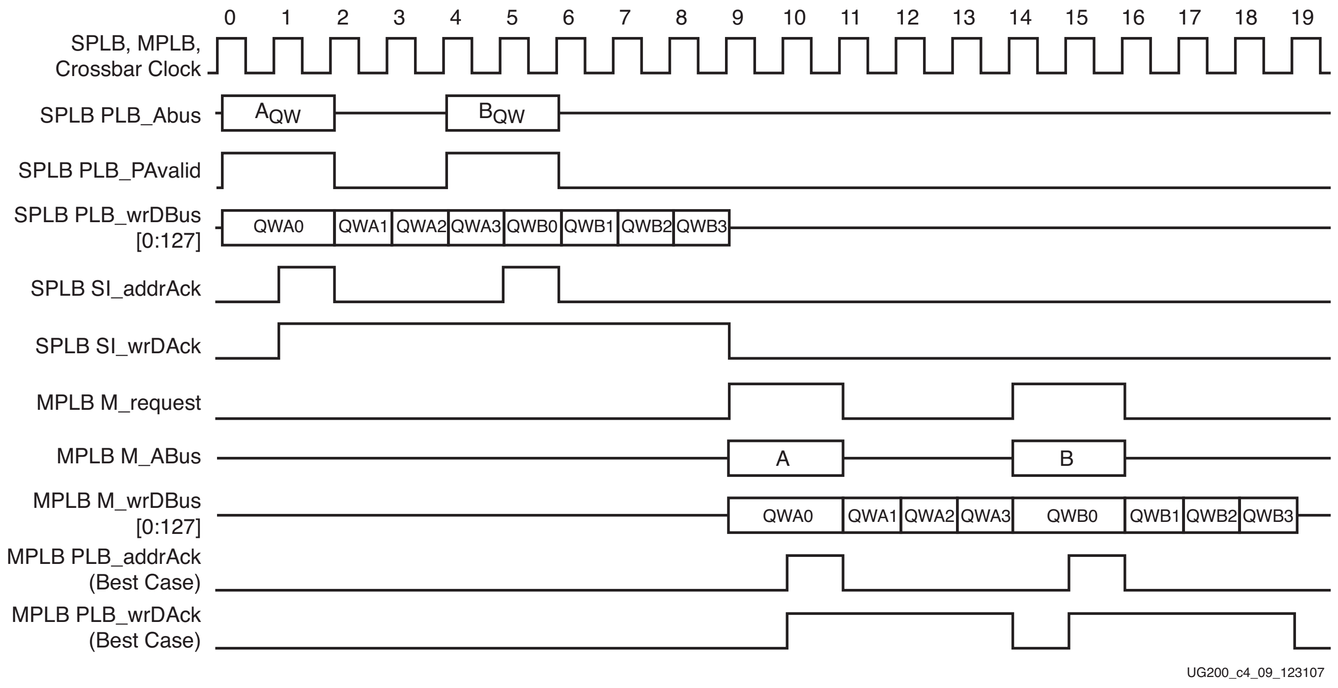


Figure 4-9: **SPLB to MPLB Back-to-Back 4-Quadword Burst Writes Between 128-Bit Devices**

Figure 4-10 shows back-to-back read burst requests, either from a master that supports address pipelining or from two masters concurrently requesting the bus. After the first request is acknowledged, the PLB arbiter asserts PLB_SAvalid. The SPLB interface always acknowledges SAvalid (provided its command queue is not already full) and latches the address and associated transfer qualifiers. Both the primary and secondary requests are forwarded to the MPLB command queue. The MPLB interface typically asserts the request for each subsequently queued command as the current read data transfer completes. This typically leaves a 2-cycle bubble on the read data bus between each successive command (assuming best-case arbiter/slave response).

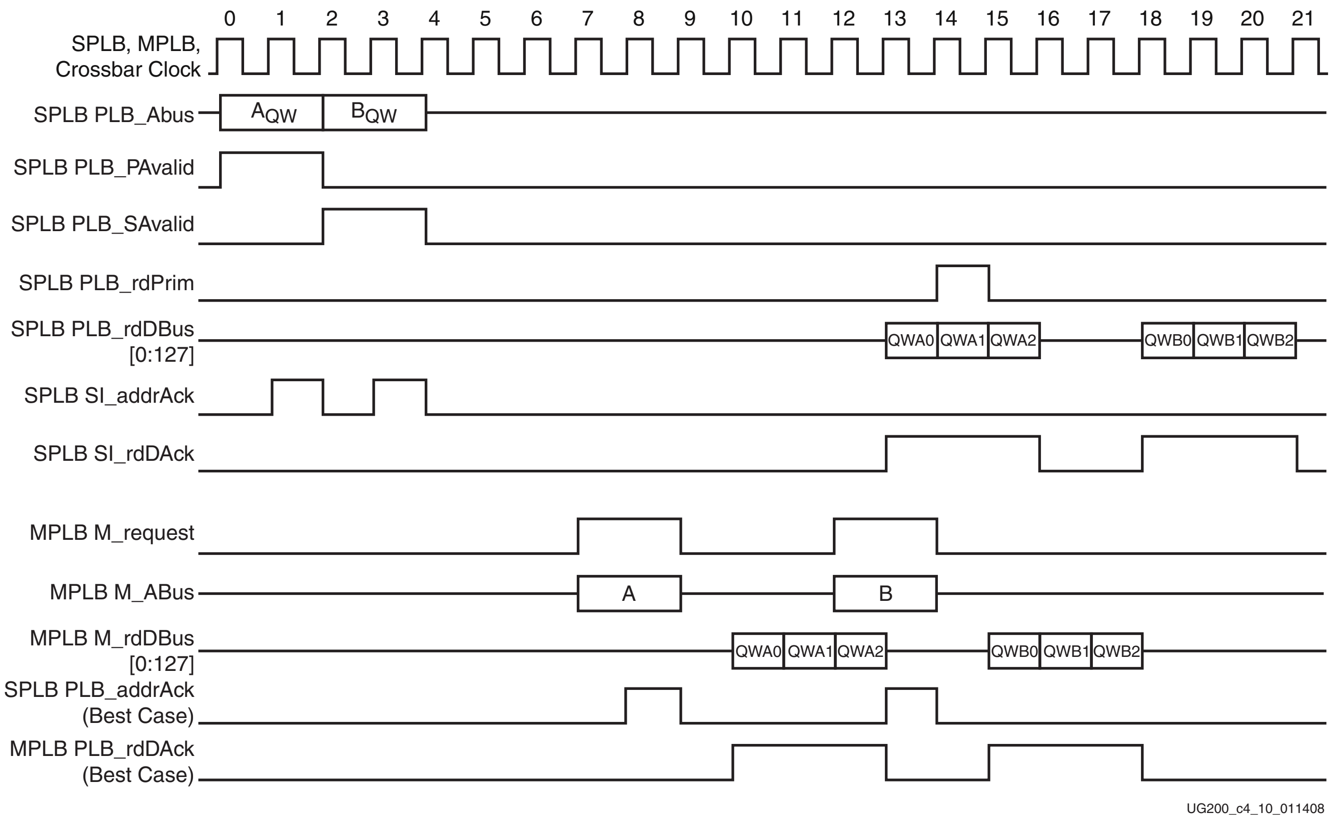


Figure 4-10: **SPLB to MPLB Pipelined 3-Quadword Burst Reads Between 128-Bit Devices (Originating Master Supports Address Pipelining)**

Figure 4-11 shows the propagation of a burst write command coming from a PLB master operating at a slower clock frequency. A 2-unit burst write takes a total of seven cycles of latency to appear on the MPLB interface. As shown in Figure 4-1, the first 3 cycles (1 + n) of pipelining occur within the SPLB interface controller, paced by the slower SPLB clock, while the final two stages in the SPLB are synchronized to the faster crossbar clock. The sixth and seventh cycles of latency occur within the crossbar and MPLB interface, respectively, also at the higher clock frequency.

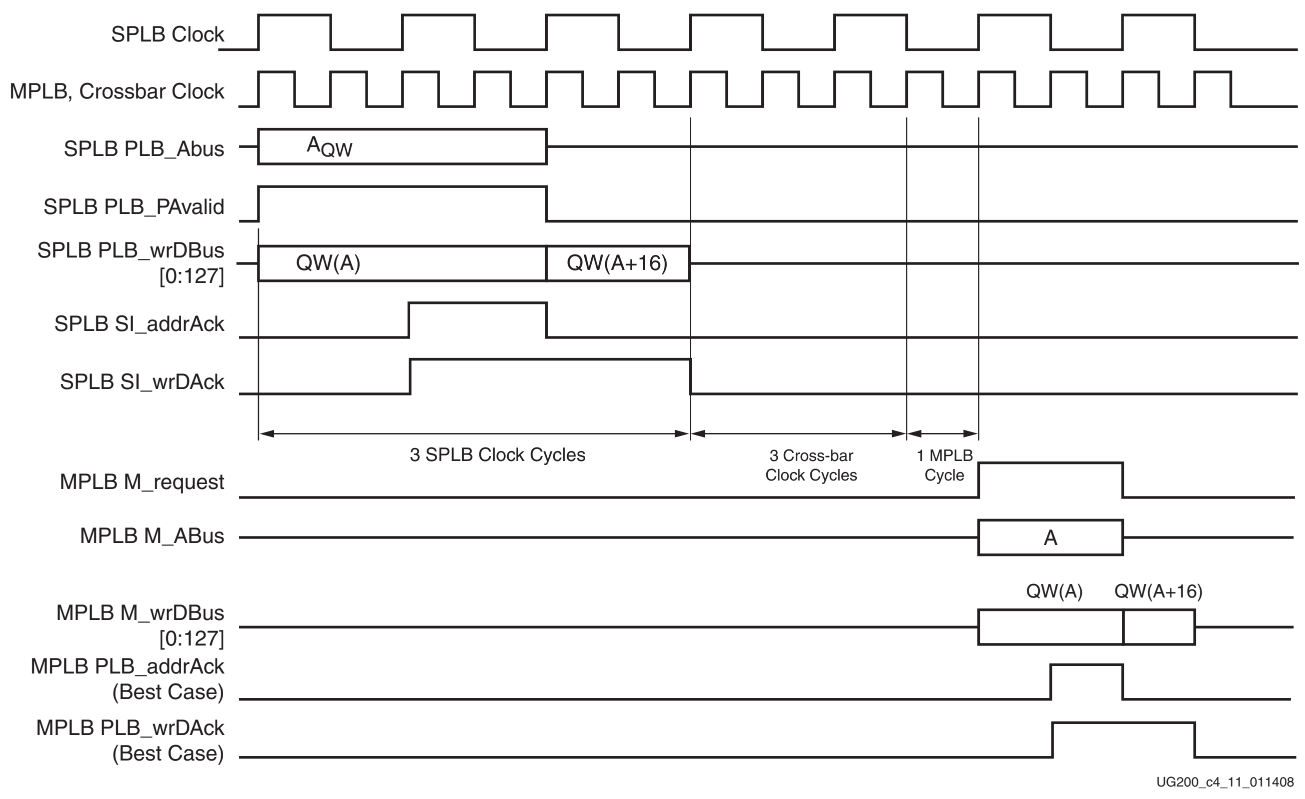


Figure 4-11: **SPLB to MPLB 2-Quadword Burst Write with 1:2 Clock Ratio**

Figure 4-12 shows the typical waveforms for a read burst originating on the SPLB that targets the MPLB interface operating at a slower clock frequency. Read bursts have a total latency of seven cycles. The first 6 cycles occur at the higher crossbar and SPLB clock frequencies, while the seventh cycle is synchronized by the slower MPLB clock. The return data path typically takes one cycle in the MPLB domain plus two cycles in the crossbar and SPLB clock domains.

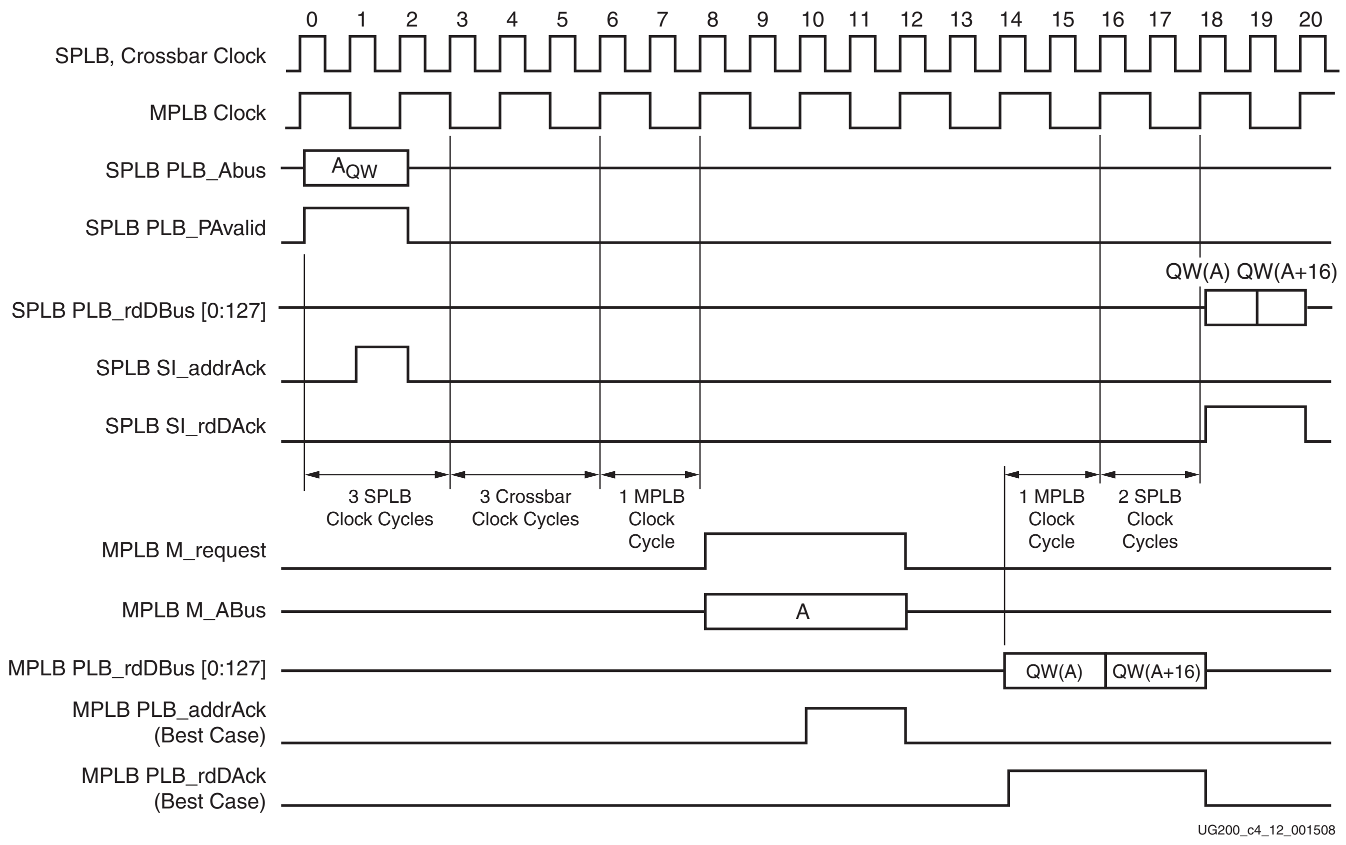


Figure 4-12: **SPLB to MPLB 2-Quadword Burst Read with 2:1 Clock Ratio**

PLB Interconnection Techniques

The crossbar in the embedded processor block provides a high-performance pathway to allow memory and other peripherals to be shared between the processor and other masters in the system. There are many ways that external masters, memories, and peripherals can be connected to the crossbar. Overall system performance is generally improved by moving away from the single shared-bus interconnect paradigm toward a network of multiple independent buses that allow data to move around the system in parallel. This section describes some of the basic PLB interconnection strategies.

Figure 4-13 depicts the simple shared-bus topology, similar to the way peripherals can be connected to a PowerPC 405 processor in earlier Virtex architectures. In this example, the "main memory" for the processor is attached to a memory controller on the PLB. The performance of this topology might be sufficient, particularly if there are no other masters in the system that need to share any of these memory or peripheral devices. Even so, access to any high-latency peripherals by the data load/store unit might occasionally stall the processor's instruction fetch.

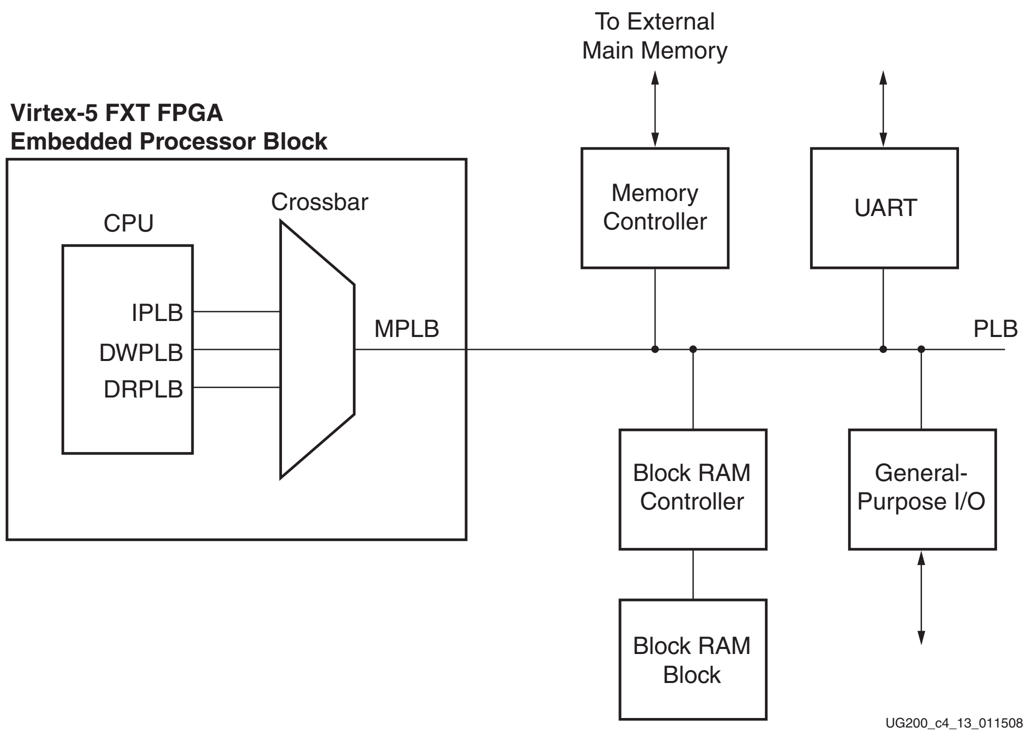


Figure 4-13: **Simple Processor-Centric Shared Bus Design**

Figure 4-14 simply replaces the PLB-based memory controller with one connected to the crossbar MCI. Overall latency to memory is slightly improved due to the elimination of PLB arbitration cycles. Because the pathways to main memory and peripherals are now independent, peripheral access can no longer interfere with instruction fetch.

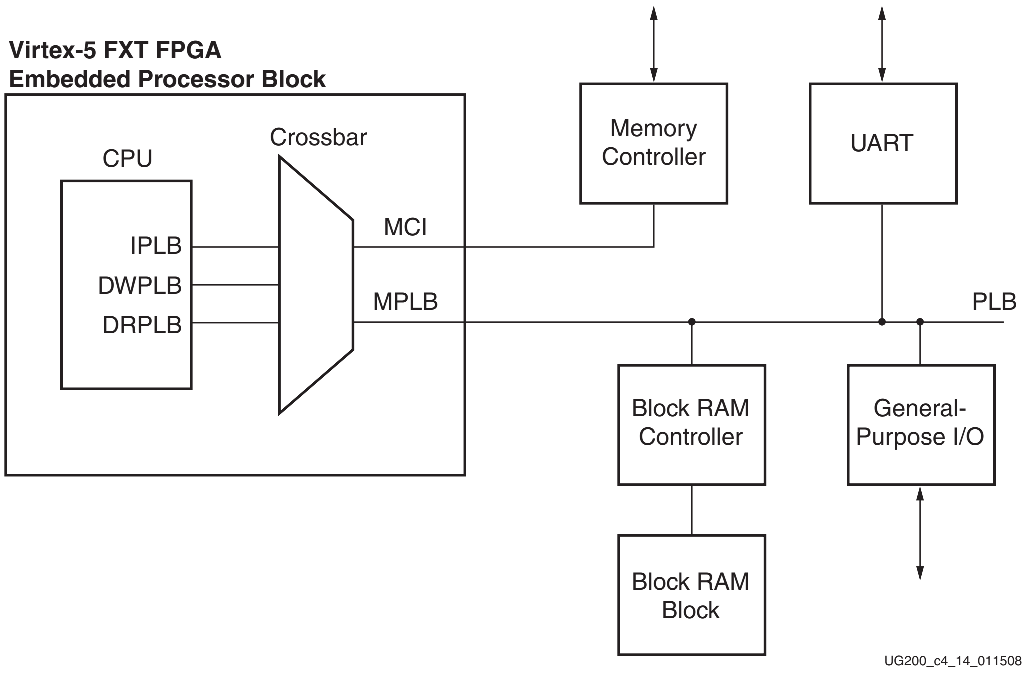


Figure 4-14: **Simple Processor-Centric Design Using Memory Controller Based Main Memory**

Figure 4-15 shows how another master device in the system can access main memory, peripherals, or both via the crossbar's SPLB 0 or SPLB 1 interface. While access to either MCI or MPLB interface is now arbitrated, access by each master to opposite interfaces can be carried out in parallel. For example, the external master can read from main memory while the processor accesses any of the peripherals via the MPLB. Also, because the hardened crossbar can operate at higher frequency than the FPGA logic, accesses to main memory by the various masters can be queued up in the crossbar to maximize memory bandwidth.

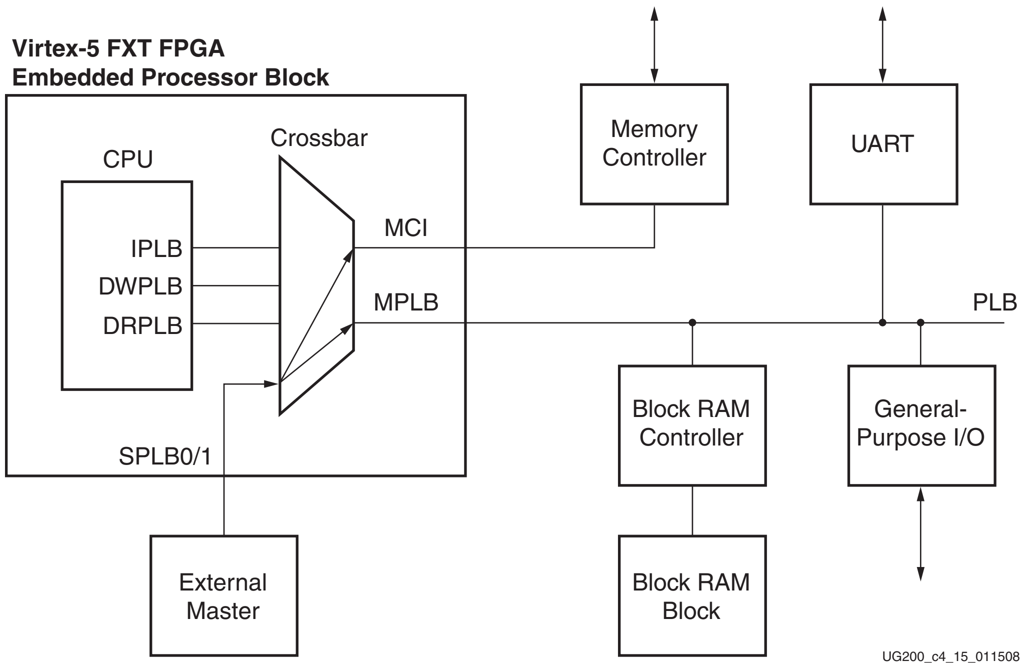


Figure 4-15: **Main Memory and Peripherals Shared Between Processor and External Master**

One form of external master is a high-speed I/O device, such as an Ethernet controller. As shown in Figure 4-16, rather than moving streams of data using PLB protocol, such devices are often connected via LocalLink channels to hardened DMA controllers inside the embedded processor block. As with the SPLB interfaces, the crossbar serves as the pathway from these DMA engines to main memory, regardless of whether main memory is connected on the MCI (as shown in Figure 4-16) or the MPLB interface (as in Figure 4-13).

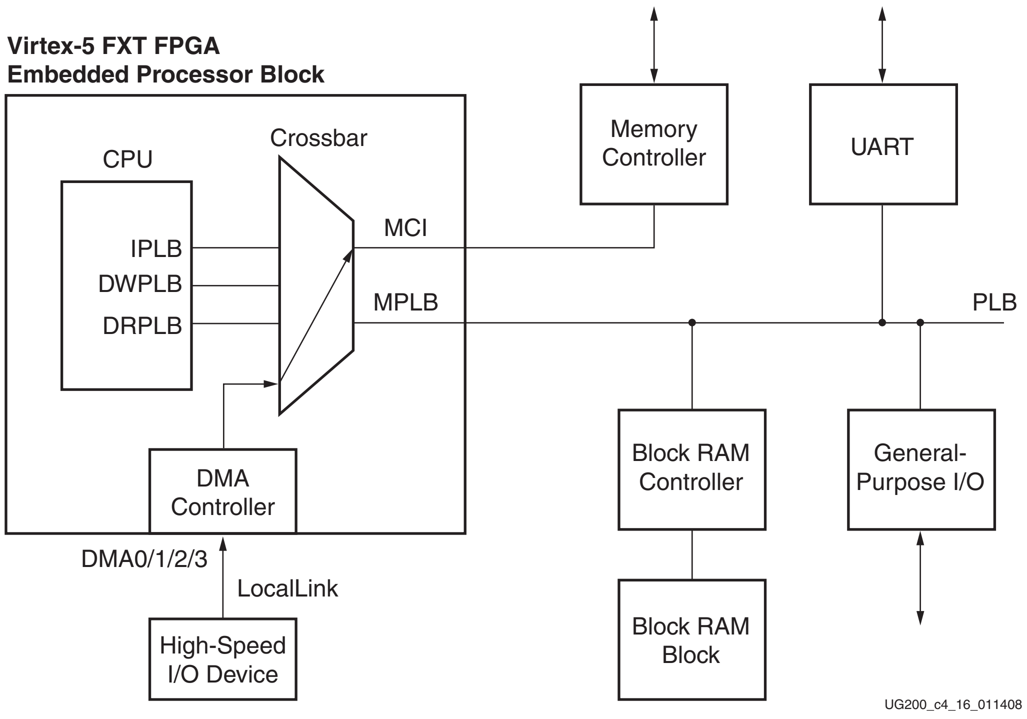


Figure 4-16: **Main Memory Shared Between Processor and DMA**

Another device that acts as both an external master and as a slave is a bridge to an external bus, such as PCI or PCI Express bus. In Figure 4-17, remote masters can access local main memory on the MCI via the crossbar SPLB 0/SPLB 1 interface. Also, the local processor can access remote slaves via the MPLB, which connects to a slave interface on the external bridge. There may also be a DMA connection to the external bridge (not shown) to accelerate locally initiated streaming transfers.

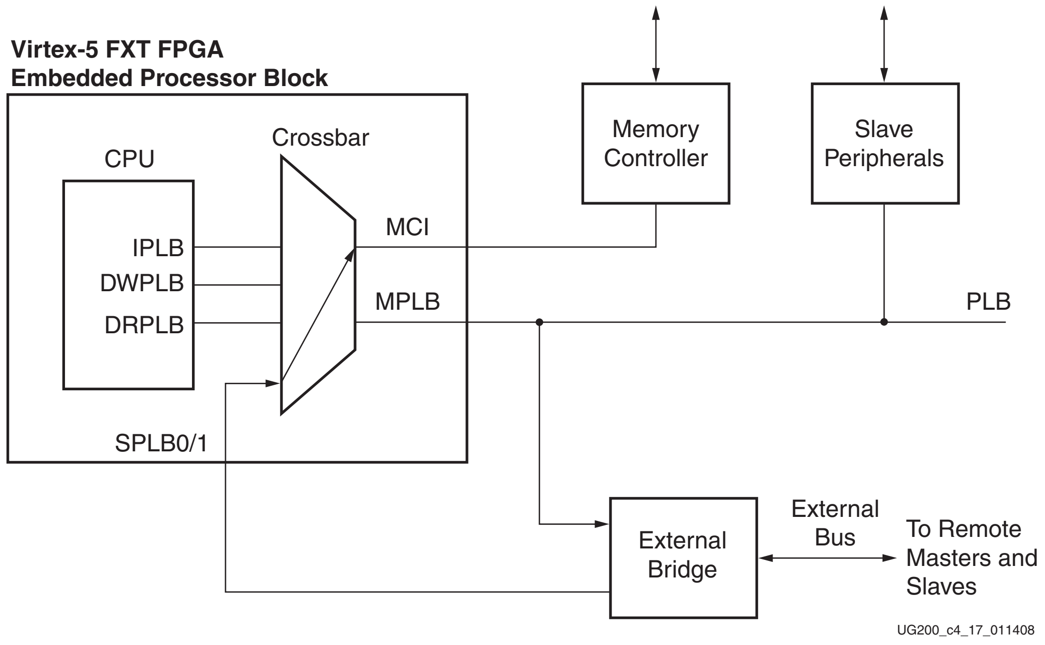


Figure 4-17: **External Bridge with Remote Access to Main Memory and Processor Access to Remote Peripherals**

If a remote master also needed to access any of the local peripherals (or PLB-based memory) connected to the MPLB interface, the crossbar can provide a pathway from SPLB 0/SPLB 1 to MPLB. However, because outbound traffic from the processor to remote slaves occurs over the MPLB and because the MPLB uses an independent PLB/arbiter than SPLB 0/SPLB 1, it is possible for such a topology to become deadlocked. For example, if the processor is granted the MPLB to access a remote slave at about the same time as a remote master requests access to the crossbar, the outbound request might be held at the external bridge until the inbound traffic completes. But if the inbound request is for access to a peripheral on the MPLB, the crossbar does not allow that transaction to proceed until the pending request from the processor completes, thus leading to deadlock.

One simple solution to avoid potential deadlock, in the case of an external bridge, is shown in Figure 4-18. Here, the same PLB is used to carry both inbound and outbound traffic, and it connects to both the MPLB and SPLB 0/SPLB 1 interfaces of the crossbar. The data traffic patterns are the same as in Figure 4-17, except that all inbound and outbound requests must first arbitrate for the same PLB before commencing, thus avoiding the risk of deadlock. However, sharing the same PLB between the processor and the external bridge master might reduce the overall throughput of the system.

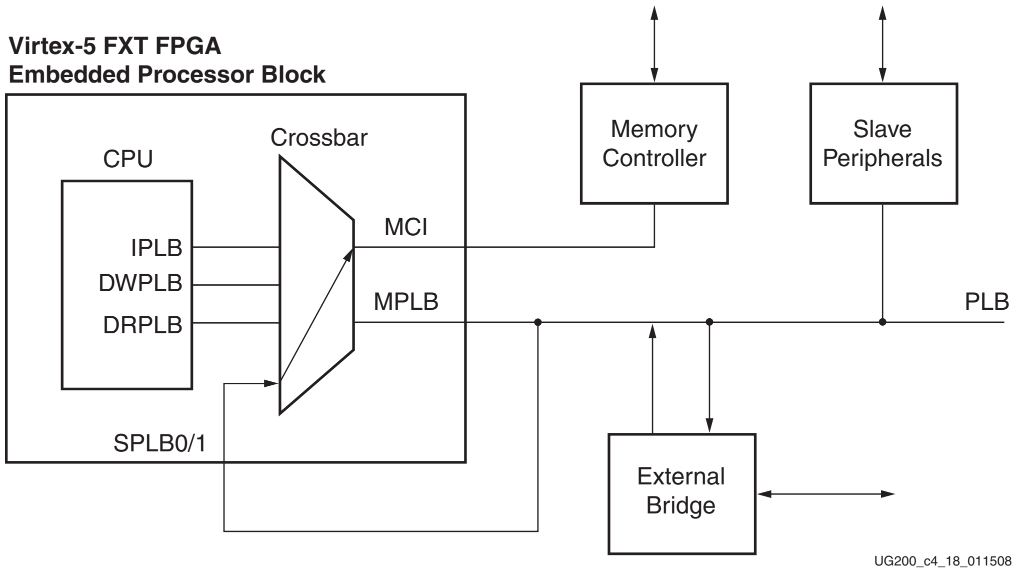


Figure 4-18: **External Bridge with Remote Access to Main Memory and Locally Shared Peripherals**

Chapter 5

Memory Controller Interface

The Memory Controller Interface (MCI) block provides a bridge between the high-speed crossbar and a soft memory controller implemented in FPGA logic. The MCI provides a simple protocol that allows the soft memory controller to run much faster because it does not need to implement the more complex and more general PLB protocol. Figure 5-1 shows a soft memory controller interfacing to a physical memory outside the Virtex-5 FPGA on one side and the MCI of the embedded processor block on the other side. This architecture allows soft memory controllers to be designed for various types of external memories such as DDR2, QDR, and so on, without building a soft PLB interface.

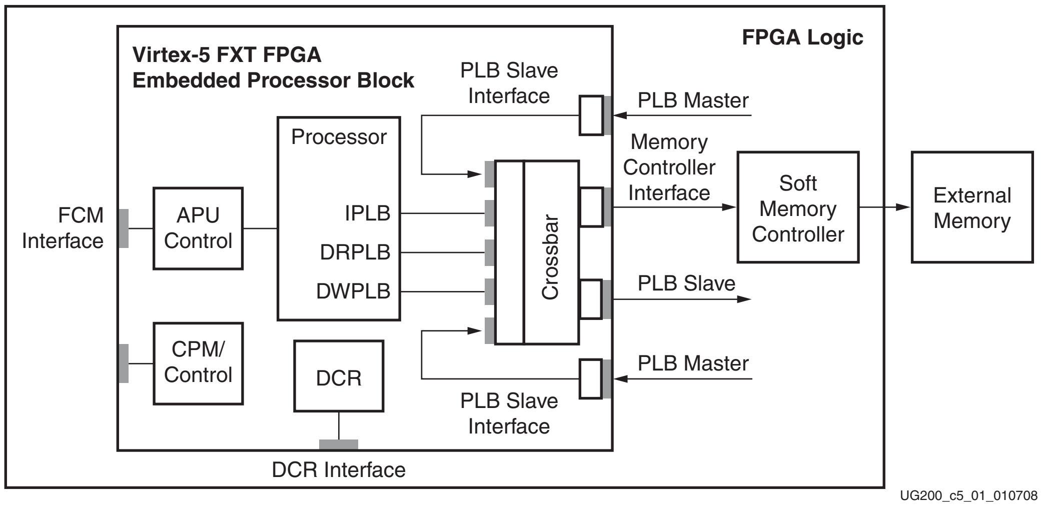


Figure 5-1: **Memory Controller Interface**

Overview

In most processor-based systems, the overall performance is highly dependent on the latency and bandwidth between the processor and memory. Large memories are typically shared by the processor and other peripherals, making memory access a bottleneck. To resolve this issue, the embedded processor block in Virtex-5 FXT FPGAs contains a high-speed memory interface connected to the crossbar to allow the processor and other high-speed peripherals to share the memory efficiently. Figure 5-1, page 131 depicts the topology of the interconnection in the processor block including the Memory Interface block.

This interface improves the performance of external memory accesses, while preserving the flexibility to use various memory types. Therefore, the physical layer of the memory controller for different types of memory, such as DDR, QDR, and SRAM, is implemented as soft logic, while the interface to the crossbar is hardened. This allows the crossbar to run at a higher frequency.

As explained in Chapter 2, "Embedded Processor Block Overview," the crossbar eliminates blocking of transactions to the memory controller while other master/slave transactions are in process. Furthermore, the MCI block can support split transactions by allowing multiple transactions to be pipelined to the memory controller at once.

The notable features of this interface are as follows:

- All transactions to the FPGA logic are in constant burst lengths set by the user through a control register (possible burst lengths are 1, 2, 4, and 8)
- Addresses to the FPGA logic are automatically incremented to accommodate for the constant burst length feature
- The width of all transactions to the FPGA logic on the MCI is defined in a user-programmable register (128, 64, and 32 bits)
- All control registers are accessible by the processor through DCR instructions or defined by the bitstream
- The MCI can operate at a clock ratio of 1:N (where N is an integer in the range [1:16]) with respect to the crossbar interconnect clock (CPMINTERCONNECTCLK)
- With a higher latency, the MCI can operate at a clock ratio of 2:3 with respect to the crossbar interconnect clock (CPMINTERCONNECTCLK)

Interface Features

The memory interface is a fast, compact, and convenient way of connecting memory to the embedded processor block in Virtex-5 FXT FPGAs. The MCI is designed to be similar to a simple FIFO interface rather than the more complicated PLB interface. The interface consists of an address bus, two data buses (one for each direction), and some control signals. All transactions to the FPGA logic are constant length, greatly simplifying the design of the soft memory controller. Every transaction requires at minimum an address, an address valid signal, and a signal (MIMCREADNOTWRITE) that indicates if the transaction is a read or a write. The MIMCADDRESSVALID signal is asserted for exactly one clock cycle for each transaction to the soft memory controller. The MCI can generate back-to-back reads and writes, and can switch from a read transaction to a write transaction on the next cycle. However, the MCI has an autohold feature that allows the next transaction to be delayed for a certain number of cycles under certain conditions as described in Table 5-1, page 135.

For write transactions, write data is presented on the write bus (MIMCWRITEDATA) WDD cycles after the MIMCADDRESSVALID signal is asserted. WDD is a user-configurable parameter whose allowed values range from 0 through 10. For burst transactions, only the first address is presented to the soft memory controller, and the soft memory controller is responsible for incrementing the address appropriately for the rest of the data beats in the burst transaction. The burst length and burst width values are set using control registers as shown in Table 5-1, and are not part of the interface signal set described in Table 5-2, page 137.

For read transactions, the MCI block expects the data and a valid MCMIREADDATAVALID signal at some point in time on the read data bus (MCMIREADDATA). The MCI does not terminate transactions that have started. The MCI block can use the byte enable signals (MIMCBYTEEANBLE[0:15]) to make the writes byte-selectable. Refer to "Signal Descriptions," page 137 for detailed descriptions of the input and output ports that constitute this interface.

Although the physical data buses are 128 bits wide, the user can optionally downsize the bus by setting the BURSTWIDTH parameter described in Table 5-1. This option allows the soft memory controller to save some area when the memory data width is less than 128 bits. For example, the user selects a 32-bit bus and the MCI has 256 bits of data to transmit, it sends out eight 32-bit back-to-back words to the fabric on MIMCWRITEDATA[0:31]. If the soft memory controller had to implement the muxes that selected different 32-bit portions of the 256-bit data, the soft memory controller would have been larger and slower. This muxing is also implemented on the read path of the MCI block, so that the soft memory controller does not have to form 128-bit words.

The soft memory controller is expected to support a single transaction type, in terms of burst length, as determined by the BURSTLENGTH parameter described in Table 5-1. The MCI converts internal PLB transactions of various burst lengths and widths into a corresponding number of the single transaction type supported by the soft memory controller. Every time the burst length is reached, a new address is generated to send to the FPGA logic. For example, assume the starting address is 0, a burst of four 128-bit words (64 bytes) is to be sent to the soft memory controller, and the MCI is configured to be 128 bits wide with a burst length of 2. The transactions on the MCI are (address 0, write 0-15), (write 16-31), (address 32, write 32-47), and (write 48-63).

The configurable BURSTLENGTH parameter allows different memory controllers with different requirements to be attached to the MCI, while keeping the logic in the soft memory controller to a minimum. For example, one memory device might only support burst lengths of 8, while another simpler memory device might only support single-word transactions (burst lengths of 1).

The MCI block takes the address directly from the crossbar and sends it to the FPGA logic, adjusting the address when required for bursts. The MCI block does not know what memory is connected to it. Therefore, if the connected memory ranges from `0x000` through `0x7FF`, when a user writes to address `0x900`, the memory at address `0x100` is overwritten. The MCI block assumes that the user is aware of this issue.

To simplify the row and bank detect logic even further, the MCI block produces two bits that tell the soft memory controller if the bank and row have changed from the previous burst. These signals are based on a mask that covers the upper 32 bits of the 36-bit address range. These signals reduce the need for slow and expensive comparison operations in the soft memory controller.

A by-product of having the bank and row conflict signals is that the MCI block knows when the soft memory controller might have to stall due to a bank or a row change. When the memory controller has to close one page and open another one, it cannot accept another transaction. In such cases, the memory controller just asserts the hold-off signal (MCMIADDRREADYTOACCEPT) to the MCI block, which stops delivering new addresses until the MCMIADDRREADYTOACCEPT signal is deasserted. Due to the time delay between the memory controller issuing the MCMIADDRREADYTOACCEPT signal and the MCI block reacting to it, an extra address or two might be released to the memory controller, causing it to overflow, unless the MCI block can predict this behavior and hold off on sending additional addresses and data. The MCI block can detect this change and internally assert the hold-off signal. Because the MCI block does not know how long to auto-assert the hold-off signal, it only asserts the hold-off signal long enough for the memory controller's MCMIADDRREADYTOACCEPT signal to start working. In other words, even though the MCI block internally asserts a hold-off signal, it relies on the memory controller to extend this signal sufficiently for the system to function.

Four events might cause the auto hold-off to occur: change of bank, change of row, change of direction from read to write or write to read, or an ECC non-complete transaction. A control register defines which of these events or combination of these events actually causes an auto hold-off. In addition, the control register defines how many cycles are inserted after a conflict occurs. Refer to "Control and Configuration," page 135 for details on how to set the control registers.

The QDR support mode allows overlapping reads and writes. In this mode, a write address with its data can be sent on the MCI immediately followed by a read address in the next cycle. Assuming that the MCI block was set for a burst length of 4, in non-QDR mode, the MCI block must wait until after the fourth cycle of the write to start a read transaction. However, in the QDR mode, certain features of the MCI block are disallowed. All conflict enable bits in the DCR should be turned off. QDR does not recognize conflicting banks, so this is no longer necessary. More details are provided in the following sections.

Another mode of operation is the Read Modify Write mode. If an agent writes to an ECC protected memory but does not write the entire protected word, the (FPGA logic based) memory controller must first read the original word, modify certain bytes with the new write data, recalculate the ECC, and finally write the new ECC protected word to physical memory. This process requires many cycles. Therefore, without this feature built into the memory controller, the following procedure has to be used. The MCI block issues a write. The memory controller detects if all byte enables are true. If not, the hold-off signals must be asserted to the MCI block and the Read-Modify-Write sequence must be started.

By the time the stall signal gets to the MCI block, the block might have issued multiple writes to the memory controller, thus overflowing its pipeline. When this feature is enabled, whenever the byte selects are not fully enabled, the MCI block institutes an auto-hold sequence, which gives the memory controller time to assert its own address-not-ready signal. Refer to "Control and Configuration," page 135 for programming information and "Timing Diagrams," page 137 for example timing diagrams. All the write data for this transaction is released to the soft memory controller before the MCI block looks at the address-not-ready signal. In other words, the MCI block finishes this transaction before deciding if it should send out the next transaction.

The PLB read data error signal (MCMIREADDATAERR), provided with the read data and data valid signals, can be asserted by the memory controller to tell the requesting master that something is wrong with the data, such as a parity error or an uncorrectable ECC error.

When the soft memory controller asserts MCMIREADDATAERR, the error is passed along with the PLB read data back to the PLB master. If the PLB master is connected to one of the SPLB interfaces or if the read transaction originates from one of the DMA controllers, the error also sets a flag in the interrupt status register (IST, DCR `0x20`) and can lead to an interrupt, if enabled. If the PLB master is the PowerPC 440 embedded processor, the read error causes a machine check exception when that data is used by the PowerPC 440 embedded processor (for data reads) or the instruction is executed by the processor (for instruction reads).

Crossbar Transactions

The MCI block interfaces with the crossbar on one side and with the soft memory controller on the other side. Each side has its own clock domain, and synchronizers allow data to cross the clock boundaries.

The MCI block receives transactions from the crossbar, which receives these transactions from a number of different devices (see Figure 2-1, page 29). These transactions can originate from either the processor (three distinct PLB masters) or from the soft FPGA logic (two masters). Memory transactions can also originate from any of the four DMA controllers built into the embedded processor block in Virtex-5 FXT FPGAs. The MCI block relies on the PLB interfaces on the crossbar inputs to simplify certain transactions, such as indeterminate bursts.

Control and Configuration

Three registers, accessible through the DCR interface, are provided to allow control and configuration of the MCI block. The default value of these registers can be set using the configuration attributes on the processor block.

MI_ROWCONFLICT_MASK [0:31] Register

This register contains the mask used to detect row conflicts from one transaction to another. This register is at DCR address `0x11`. The 32 bits in this register correspond to the higher order 32 bits of the 36-bit address generated by the MCI. A 1 in any bit position identifies that bit as a row address bit. For example, if bits 8:20 are set to 1, the MIMCROWCONFLICT signal is set to 1 if the corresponding bits of MIMCADDRESS change between the previous instruction sent to the soft memory controller and this instruction. The default value of this register is 0.

MI_BANKCONFLICT_MASK [0:31] Register

This register contains the mask used to detect bank conflicts from one transaction to another. This register is at DCR address `0x12`. The 32 bits in this register correspond to the higher order 32 bits of the 36-bit address generated by the MCI. A 1 in any bit position identifies that bit as a bank address bit. For example, if bits 4:7 are set to 1, the MIMCBANKCONFLICT signal is set to 1 if the corresponding bits of MIMCADDRESS change between the previous instruction sent to the soft memory controller and this instruction. The default value of this register is 0.

MI_CONTROL [0:31] Register

This control register is at DCR address `0x10`. Table 5-1 describes the meaning of each bit in the register.

Table 5-1: **Bit Descriptions for the MI_CONTROL Register**

Bit	Name	Default Values	Description
[0]	enable	0	• 1: The MCI is enabled. The PLB and DMA masters can access the soft memory controller through the crossbar. • 0: The MCI is disabled, and any attempt to access the MCI through the crossbar will fail.
[1]	Rowconflictholdenable	0	If there is a change between the row from the current address and the past address, setting this bit causes the MCI block to wait Autoholdduration number of cycles before starting up the next instruction.
[2]	Bankconflictholdenable	0	If there is a change between the bank from the current address and the past address, setting this bit causes the MCI block to wait Autoholdduration number of cycles before starting up the next instruction.
[3]	Directionconflictholdenable	0	If there is a change of direction between the current address and the past address (from reads to writes and writes to reads), setting this bit causes the MCI block to wait Autoholdduration number of cycles before starting up the next instruction.

Table 5-1: **Bit Descriptions for the MI_CONTROL Register *(Cont'd)***

Bit	Name	Default Values	Description
[4:5]	Autoholdduration	`00`	This field tells the MCI block how long to hold off when there is a triggering event causing an autohold. • `00`: 2 cycles • `01`: 3 cycles • `10`: 4 cycles • `11`: 5 cycles
[6]	2:3 Clock Ratio mode	`0`	Clock ratio mode: • `0`: Integer ratio of the MCI clock to the embedded processor block interconnect clock (CPMINTERCONNECTCLK) • `1`: Fractional ratio of the MCI clock to the embedded processor block interconnect clock (CPMINTERCONNECTCLK) (3/2)
[7]	overlaprdwr	`0`	If this bit is set, a read transaction does not always block the next write transaction from going out. If this bit is not set, after every read or write, the amount of time for that burst will transpire before the next transaction is issued. This bit should be enabled for QDR.
[8:9]	Burstwidth	`00`	Data per clock cycle: • `00`: Burst width = 128 • `01`: Burst width = 64 • `10`: Reserved • `11`: Burst width = 32
[10:11]	Burstlength	`00`	Burst length: • `00`: Burst length = 1 • `01`: Burst length = 2 • `10`: Burst length = 4 • `11`: Burst length = 8
[12:15]	Write Data Delay (WDD)	`0000`	Values 0 through 10 are valid delays in terms of clock cycles. Values 11 through 15 are reserved.
16	RMW	`0`	When this bit is set, if all the byte enables for a write are not enabled for this transaction, the MCI waits for a number of cycles determined by Autoholdduration before starting the next transaction.
[17:23]	Reserved	`0000000`	These bits are reserved
24	PLB Priority Enable	`1`	• `0`: First level arbitration is disabled for the PLB masters trying to access the MCI through the crossbar. • `1`: First level arbitration is enabled among the PLB masters trying to access the MCI through the crossbar See "Arbitration" in Chapter 3 for more information.
[25:27]	Reserved	`000`	These bits are reserved
[28]	Pipelined Read Enable	`1`	• `0`: The crossbar does not accept a new read command until the current read command completes. • `1`: The crossbar accepts read commands destined for the MCI while the current read operation is still in progress.

Table 5-1: **Bit Descriptions for the MI_CONTROL Register *(Cont'd)***

Bit	Name	Default Values	Description
[29]	Pipelined Write Enable	1	• 0: The crossbar does not accept a new write command until the current write command completes. • 1: The crossbar accepts write commands destined for the MCI while the current write operation is still in progress.
[30]	Reserved	1	Reserved
[31]	Reserved	1	Reserved

Signal Descriptions

Table 5-2 describes the MCI signals.

Table 5-2: **Memory Controller Interface Signals**

Signal	Dir	Description
MIMCREADNOTWRITE	O	Transaction type: • 0: Write • 1: Read
MIMCADDRESS[0:35]	O	Byte address
MIMCADDRESSVALID	O	This signal is the valid bit associated with MIMCADDRESS.
MIMCWRITEDATA[0:127]	O	Write data
MIMCWRITEDATAVALID	O	This signal indicates if the data on MIMCWRITEDATA is valid.
MIMCBYTEENABLE[0:15]	O	This bus determines which bytes of MIMCWRITEDATA are to be written to the RAM.
MIMCBANKCONFLICT	O	The soft memory controller uses this signal to determine if this address is in the same bank as the previous instruction.
MIMCROWCONFLICT	O	The soft memory controller uses this signal to determine if this address is in the same row as the previous instruction.
CPMMCCLK	I	MCI clock
MCMIREADDATA[0:127]	I	Read data
MCMIREADDATAVALID	I	The soft memory controller asserts this signal to let the Memory Controller Interface block know that the data presented on MCMIREADDATA is valid.
MCMIREADDATAERR	I	The soft memory controller asserts this signal to indicate something is wrong with the read data, possibly a parity or ECC error that was detected by the memory controller.
MCMIADDRREADYTOACCEPT	I	Whenever the soft memory controller is ready to accept another complete transaction from the MCI, it asserts this signal.

Timing Diagrams

The diagrams in this section show how this interface is used and the relationship between the signals. The diagrams show actual MCI signals, but PLB signal names can apply to any one of the PLB connections to the crossbar that can drive the MCI. Actual signal latencies across the crossbar are not shown. The conventions for bus values are as follows:

- A = address
- WR= write data
- RD = read data

The first digit after the letter represents the cycle transaction from the PLB. The first digit is followed by a period. The last set of digits represents blocks of data. For instance, if the PLB is transmitting two 128-bit quantities, it sends out WRX.0 and WRX.1.

Figure 5-2 shows major activity for a write request through all the major blocks of the hard interconnection. The operation starts at the processor, goes through the crossbar, the memory interface, the memory controller, and the actual memory (DDR). When the transaction leaves the MCI, the number of cycles depends on the type of memory. Therefore line breaks are used to indicate the variable cycle times.

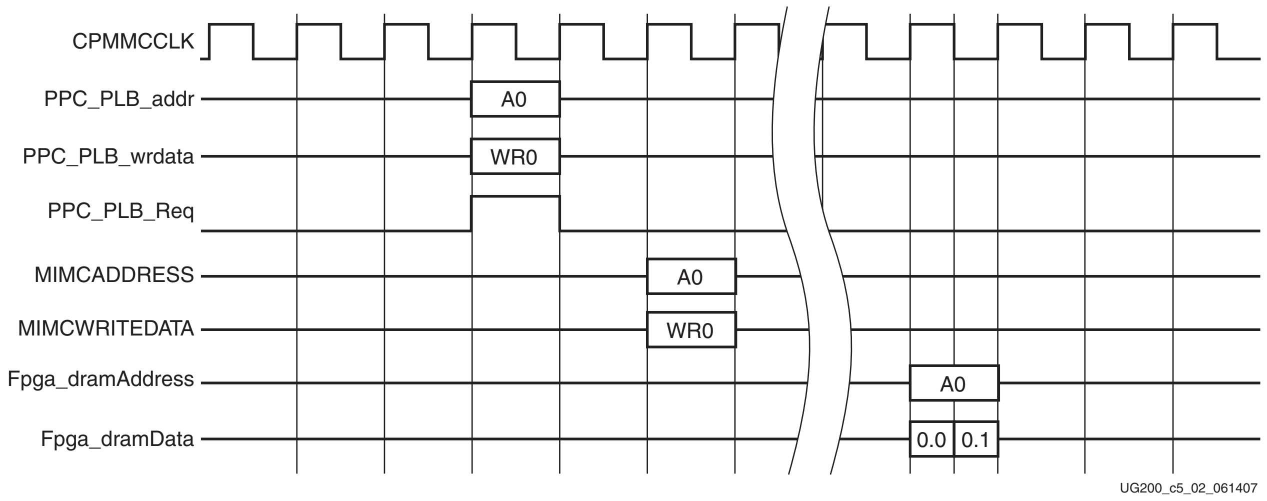


Figure 5-2: **System-Level Timing Diagram: Write**

Figure 5-3 shows the key activities for a read request through all major blocks of the hard interconnection. The request starts at the processor, goes through the crossbar, the memory interface, the memory controller, the actual memory (DDR), and then back through those blocks. When the transaction leaves the MCI, the number of cycles depends on the type of memory. Therefore line breaks are used to represent the variable cycle times.

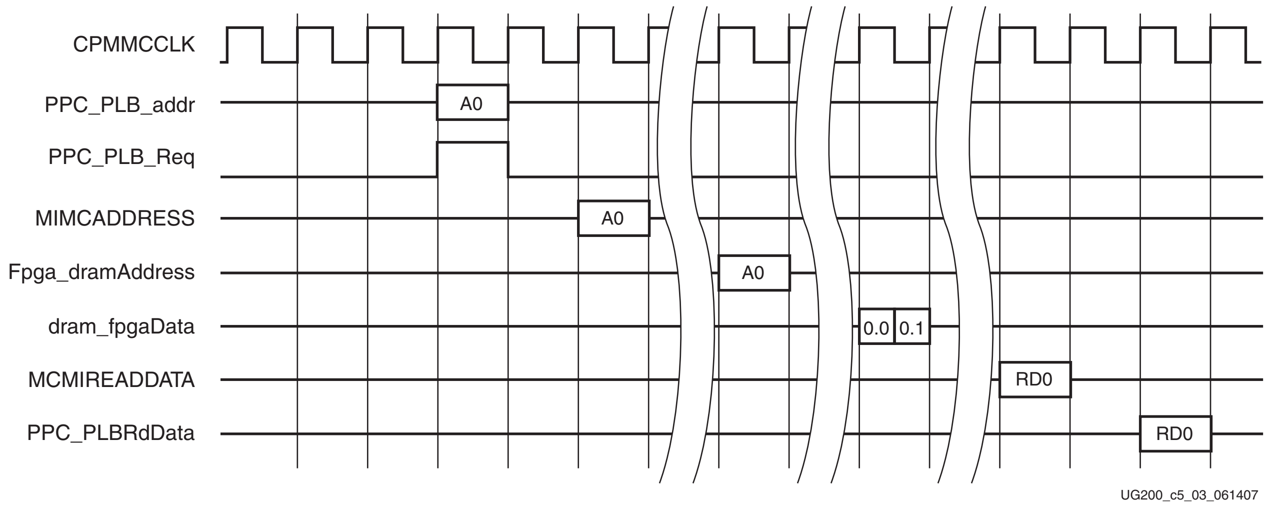


Figure 5-3: **System-Level Timing Diagram: Read**

Figure 5-4 shows a PLB burst sent through the MCI block. The address and data come out to the memory controller just as they arrived at the PLB interface.

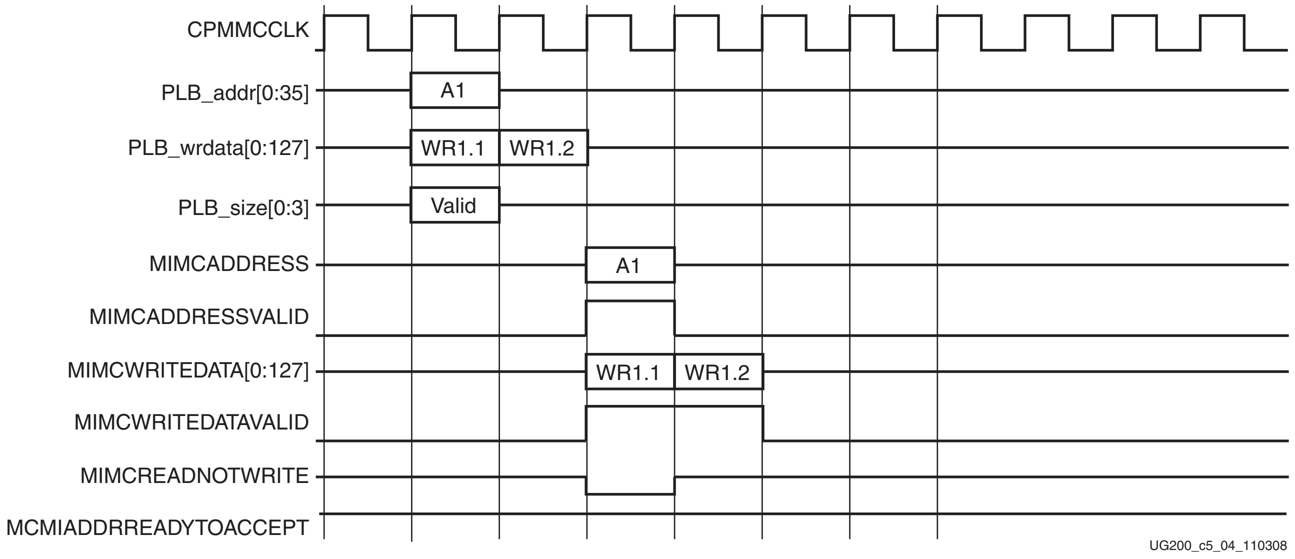


Figure 5-4: **Burst Transfer 1 (Memory Controller Interface = 128 Bits, Burst = 2, PLB = Cache Line)**

Figure 5-5 shows how a burst transfer from the PLB can get split into two separate transactions on the memory interface. The first line of data (A1 and WR1.1) is sent out without many changes. In the second burst to the memory controller, the MCI block creates a new address (A1.1) to go with the second set of data (WR1.2). Address A1.1 is incremented as appropriate.

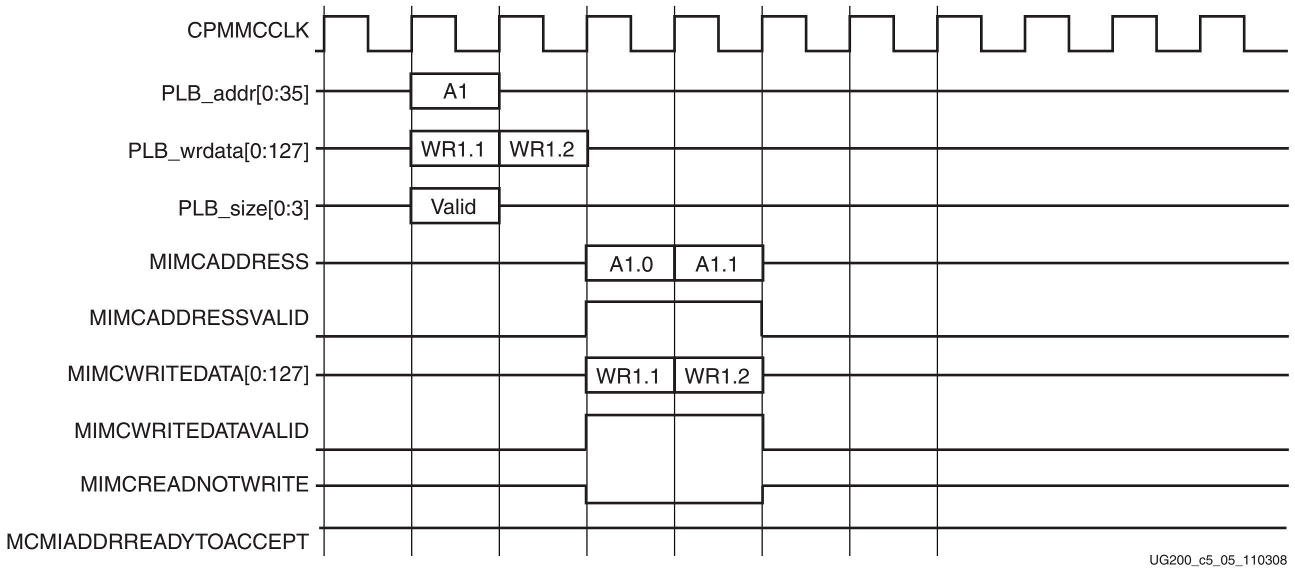


Figure 5-5: **Burst Transfer 2 (Memory Controller Interface = 128 Bits, Burst = 1, PLB = Cache Line)**

Figure 5-6 shows how the data can get broken into smaller blocks on the MCI. The PLB sends the cache line in two 128-bit beats. Beat 1 has blocks 0 and 1 while Beat 2 has blocks 2 and 3, where each block is 64 bits. When the address comes out of the FIFO, the control logic determines that not only does it have to create two separate bursts out of the original PLB transaction, but each of the new transactions has to have its data broken into smaller blocks. As can been seen on MIMCWRITEDATA, each 64-bit quantity comes out in its own cycle. This is helpful if the memory controller is connected 32-bit DDR DRAM.

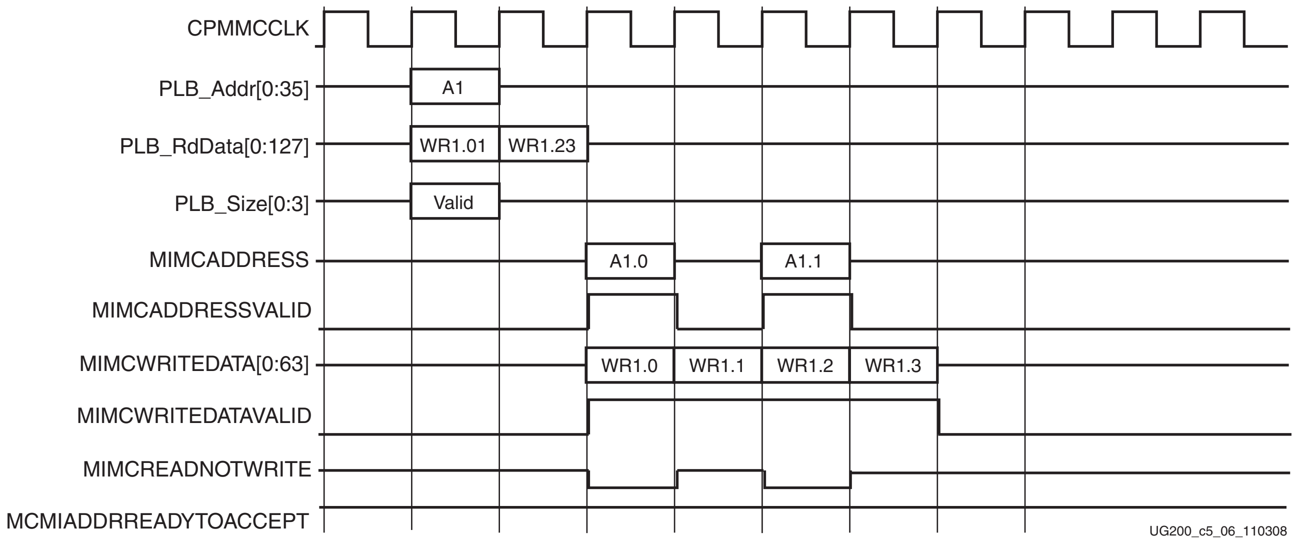


Figure 5-6: **Data Translation (Memory Controller Interface = 64 Bits, Burst = 2, PLB = Cache Line)**

Figure 5-7 shows the same basic data translation as Figure 5-6, except when the MCI is ready to send the data to the soft memory controller, the soft memory controller has not asserted MCMIADDRREADYTOACCEPT. The MCI waits for MCMIADDRREADYTOACCEPT to go High before starting the transaction. When the signal goes High, the MCI starts sending the new data to the soft memory controller three cycles later. Because the MCI is in 64 x 2 mode, only bits [0:63] of the data bus and bits [0:7] of the byte enable should be looked at; the rest of the bits are undefined.

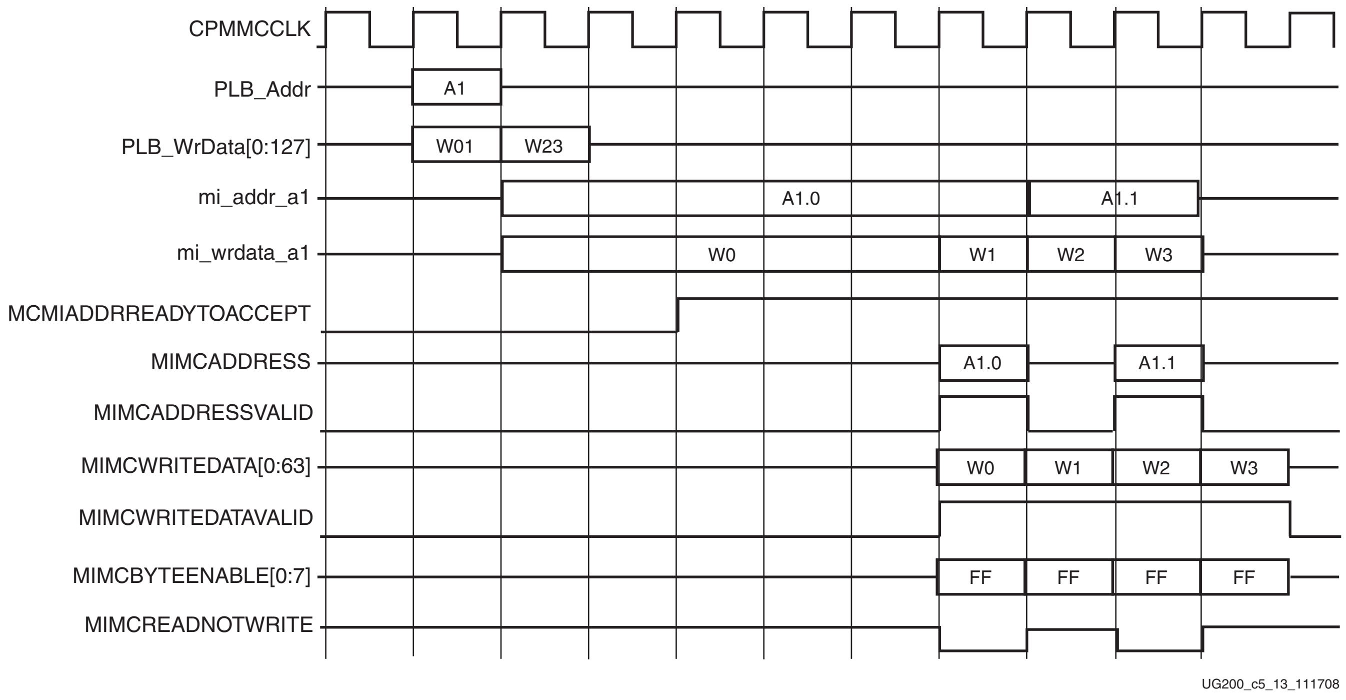


Figure 5-7: **Memory Controller Not Ready to Accept**

Figure 5-8 shows two transactions coming from the PLB. The first transaction is two beats of 128 bits of write data, and the second transaction is one beat of 128 bits of data. Because MCI parameters are set as Burstlength = 4 and Burstwidth = 32 (4 x 32), a new address is required for every 128 bits of data. The first PLB transaction is broken up to 2 bursts of 4 beats, each beat being 32 bits (2 x 4 x 32 = 256 bits of data). Addresses A1.0 and A1.1 are from the PLB address A1. The second transaction from the PLB is only 128 bits, so only one address is generated on the MCI.

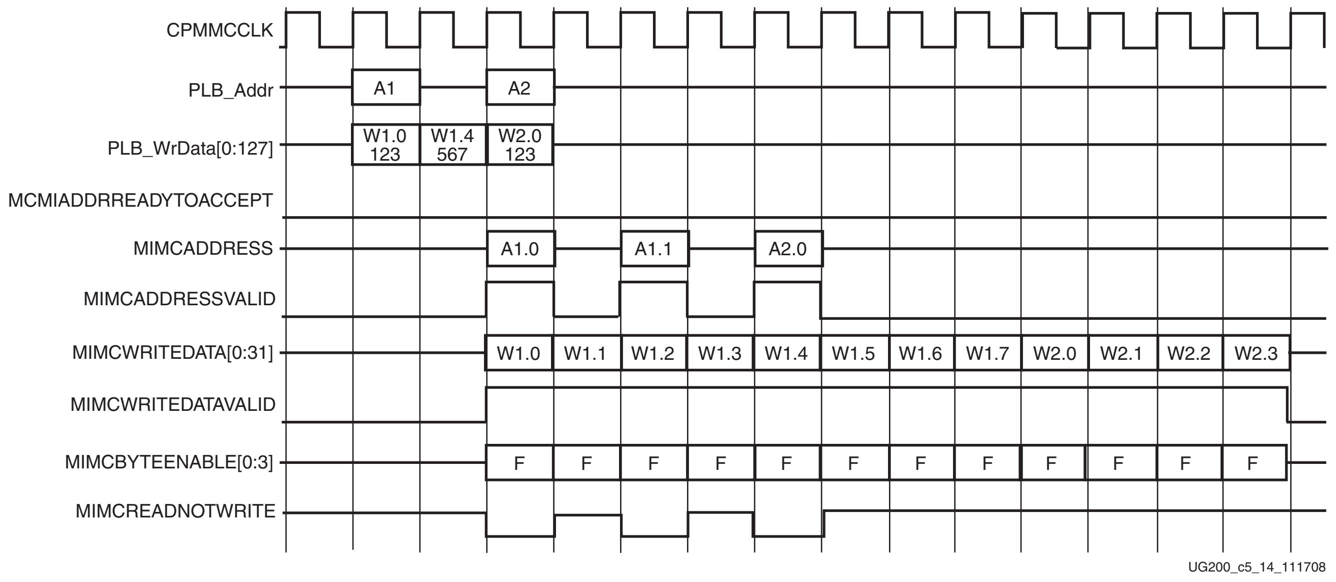


Figure 5-8: **Burst Length Set to 4x32**

Figure 5-9 shows two transactions coming from the PLB. The first one is 2 beats of 128 bits of write data, and the second one is 1 beat of 128 bits of data. The first PLB transaction is matched in size and length to the MI, so it goes out exactly as it came in (2 x 128). The second transaction is only 128 bits wide, so it has to have 128 bits of data padded (all MI transactions are 2 x 128). In this case in cycle 8, the write data is still valid, however, the byte enables are set to zero to make sure the data is not actually written into any real memory.

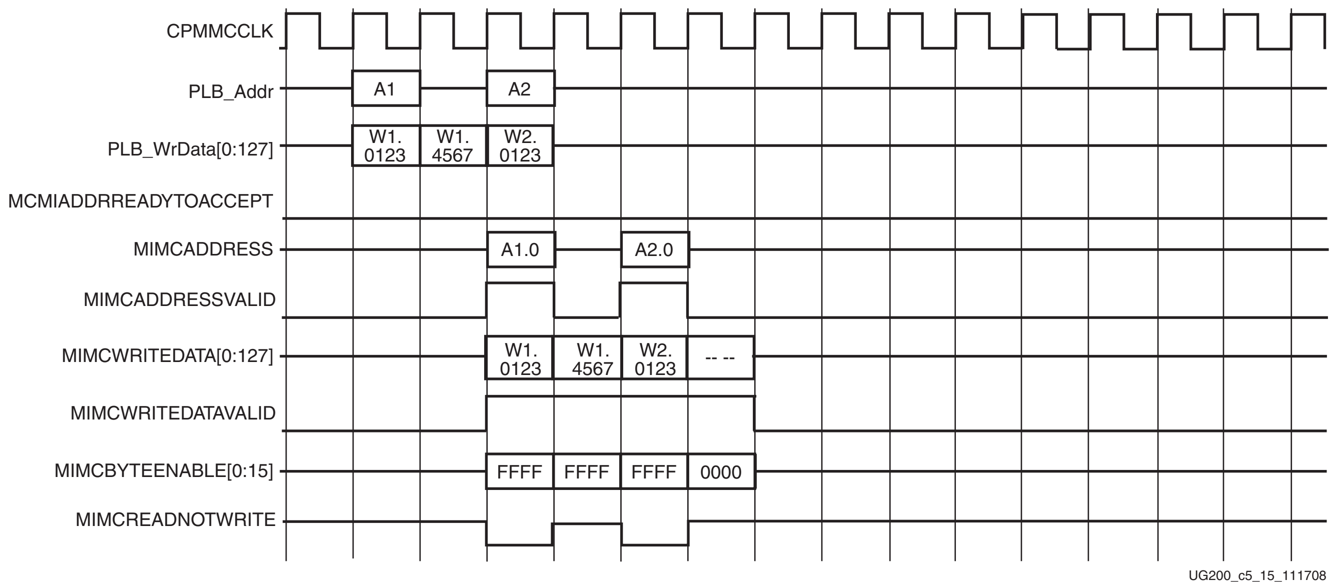


Figure 5-9: **Burst Length Set to 2x128**

Figure 5-10 shows the QDR operation. The PLB pushes a write, a read, and another write to the MCI block. Assuming that none of these transactions overlap in address space, the MCI block splits them up into two transactions each (the MCI block transactions are 2 x 64, while the PLB transactions are 2 x 128). Just after the first burst of the first write goes out, a read can occur before the next burst from that first transaction (R1.0 is between W1.0 and W1.2). The data is eventually returned to the MCI block as if this is not a QDR transaction.

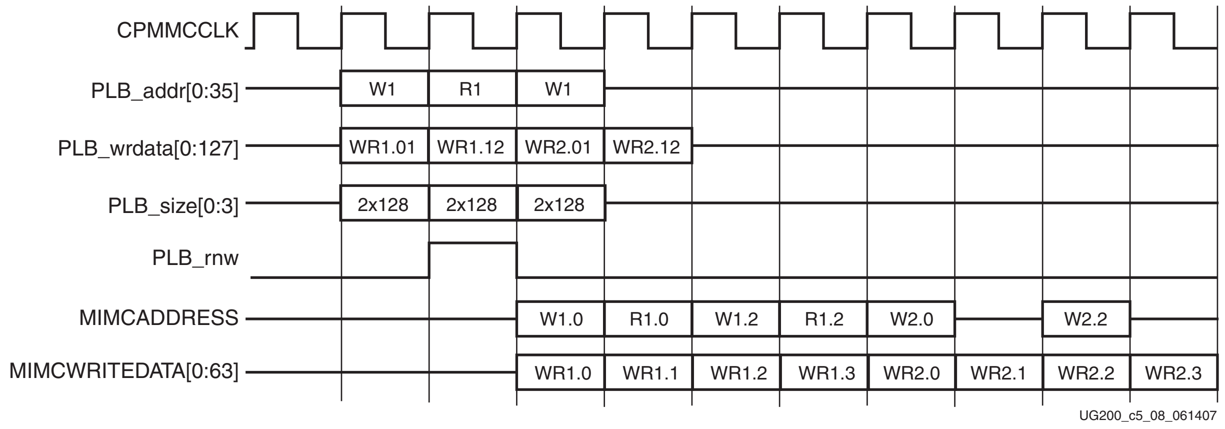


Figure 5-10: **QDR Mode (Memory Controller Interface Block Transactions = 2 x 64)**

In Figure 5-11:

- Autoholdduration = `2'b10` (4 cycles)
- WDD = 2
- Burst length = 2

Figure 5-11 shows when the MCMIADDRREADYTOACCEPT signal can be toggled in relation to the MIMCADDRESSVALID signal.

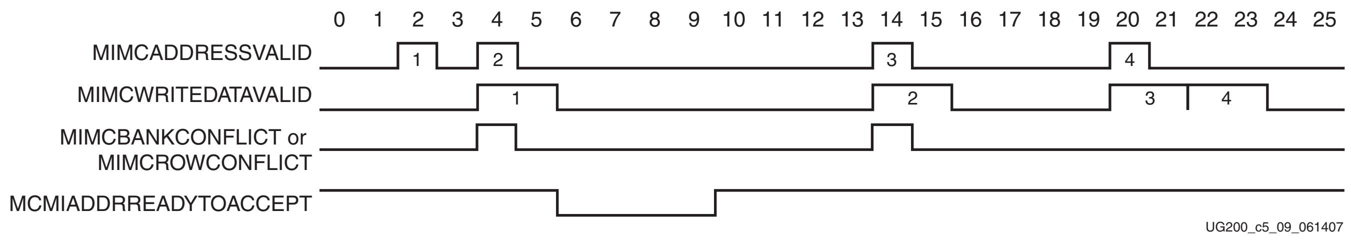


Figure 5-11: **Autohold and MCMIADDRREADYTOACCEPT**

In cycle 2, a normal, nonconflicted transaction occurs. Because WDD is set to 2, the data associated with that address occurs in cycles 4 and 5. (The burst length is set to 2, so there are two beats of data.) In cycle 4, an address is released that has a conflict. Due to the conflict, an autohold is asserted, lasting four cycles.

Because the conflict signal is asserted, the memory controller chooses to lower the MCMIADDREADYTOACCEPT signal two cycles later (cycle 6). The MCI block waits for the memory controller to release the signal before sending transactions again.

Transaction 3 occurs in cycle 14. There is also a conflict signal, however, the memory controller chooses not to toggle the MCMIADDREADYTOACCEPT signal. Therefore, the interface presents its next address six cycles later (four cycles for autohold off plus two cycles for the burst length).

In cycle 20, transaction 4 begins. Because it is not a conflict, its data is presented two cycles after the addressvalid is shown. Remember that WDD equals 2.

In Figure 5-12:

- Autoholdduration = `2'b00` (two cycles)
- WDD = 2
- Burst length = 2

Figure 5-12 shows how the autoholdduration value can be set to zero. If the soft memory controller has a combinational path from the conflict bits to the MCMIADDREADYTOACCEPT signal, in theory the memory controller could react instantaneously to the assertion of the conflict. This operation is not recommended in real systems due to timing issues; however, it provides insight on how the autoholdduration value affects the next transaction.

Transaction 1 starts in cycle 2, the same as Figure 5-6.

Transaction 2 starts the same in cycle 4; however, the memory controller deasserts the MCMIADDREADYTOACCEPT signal in the same cycle. Because autoholdduration is set to `00`, the ready_window is shown much closer to the addressvalid signal. Because of the earlier start, the MCMIADDREADYTOACCEPT signal can be deasserted earlier, and the next transaction can begin in cycle 13. (The length of the MCMIADDREADYTOACCEPT signal being deasserted in Figure 5-7 is five cycles while in Figure 5-6 it is four cycles.)

The space between cycles 3 and 4 is now only four cycles (autoholdduration is set to 0, which means two cycles of delay plus the two cycles of burst length).

Transaction 4 is identical, except that it starts earlier.

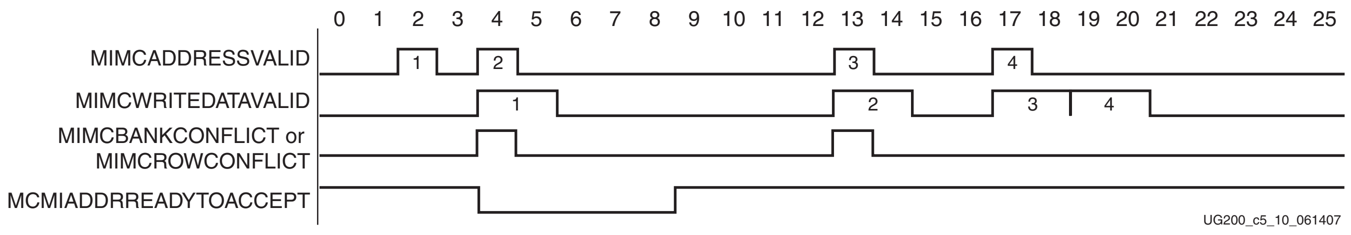


Figure 5-12: **Small Autohold**

In Figure 5-13, Autoholdduration = 2 internal cycles (0 external cycles). Figure 5-13 shows a system that is expecting a combinatorial path between conflict and MCMIADDRREADYTOACCEPT; however, the MCMIADDRREADYTOACCEPT signal goes Low in the cycle after the conflict (see time slice 4-5).

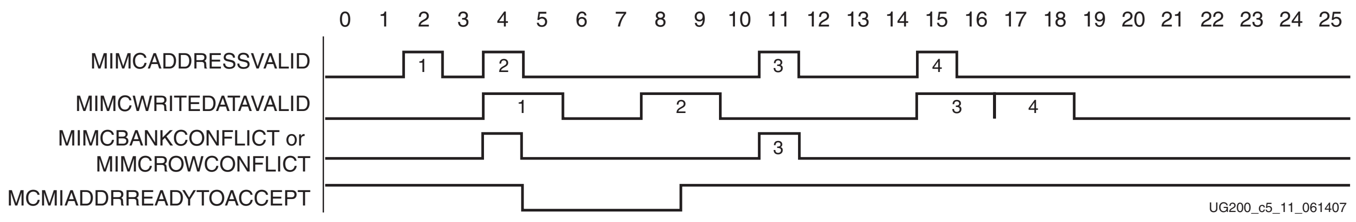


Figure 5-13: **Missed Ready to Accept**

In Figure 5-14:

- Autoholdduration = `2'b10`
- WDD = 2
- Burst length = 2
- Burst width = 32 bits

Figure 5-14 shows what happens when the RMW bit is enabled in the control register. The data width is only 32 bits, so only 4 bits of the byte enable are used. An "F" means that all bytes are being written, and the "4" in cycle 6 shows that only one byte is being written.

In cycle 2, a normal transaction is issued, whose data phase is complete at the end of cycle 5.

In cycle 4, a transaction begins, but the first beat only has one byte being written. Therefore autohold is turned on. Two cycles after the last byte enable for that transaction is shown, the memory controller drives the MCMIADDRREADYTOACCEPT signal Low, which the MCI block uses to stop giving new transactions. After the memory controller releases the MCMIADDRREADYTOACCEPT signal, the MCI block starts up new transactions, as indicated in cycle 13.

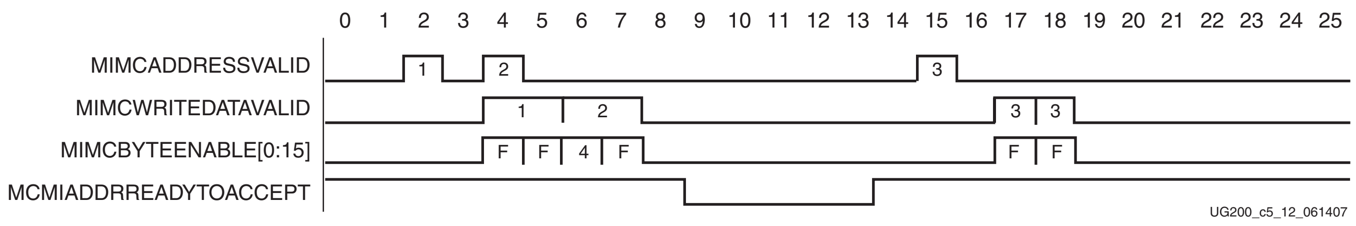


Figure 5-14: **RMW Enabled**

Board Layout Considerations

FPGAs afford a great deal of pinout flexibility; however, to maximize the performance of the PowerPC 440 processor to external DDR2 RAM, specific pin assignments must be used. To plan for migration between devices/packages, use these recommended pin assignments with Virtex-5 LXT or SXT devices in the same package. The recommended pinouts are available as UCFs in the data directory of the memory controllers that support the PowerPC 440 processors.

Chapter 6

Reset, Clock, and Power Management Interfaces

Overview

This chapter describes the reset, clock, and power management interfaces to the embedded processor block in Virtex-5 FPGAs. These interfaces feature simplified signal timing and behavior for the user. The embedded processor block internally performs clock, reset, and control signal conditioning and synchronization to meet the needs of the processor and the subsystem within the embedded processor block. Logic inside the embedded processor block also helps ensure the block is properly reset upon FPGA configuration, and that unused logic is placed in a stable state.

Reset, Clock, and Power Management Interface

Table 6-1 describes the signals in the reset, clock, and power management interface of the embedded processor block.

Table 6-1: **Processor Block Reset, Clock, and Power Management Interface Signals**

Signal Name	Direction	Default Value	Clock	Function
C440RSTCHIPRESETREQ	Output	-	CPMC440CLK	Indicates the processor is requesting a reset for itself and the processor block.
C440RSTCORERESETREQ	Output	-	CPMC440CLK	Indicates the processor is requesting a reset for just itself.
C440RSTSYSTEMRESETREQ	Output	-	CPMC440CLK	Indicates the processor is requesting a reset for the system.
CPMC440CLK	Input	1		Main processor clock.
CPMC440CLKEN	Input	1		Clock enable for the processor. A value of `1` specifies an active clock to the processor. A value of `0` disables the clock. Disabling the clock reduces dynamic power consumption.
CPMC440CORECLOCKINACTIVE	Input	0	Static	When asserted, this signal indicates that the clock to the processor is disabled. This signal is sent by an external CPM to the processor to allow a debugger to read the status of the clock.

Table 6-1: **Processor Block Reset, Clock, and Power Management Interface Signals *(Cont'd)***

Signal Name	Direction	Default Value	Clock	Function
CPMC440TIMERCLOCK	Input	1	Async (Sampled by CPU clock)	Controls the frequency of the processor timers (Time Base, Watchdog, FIT, and DEC). The frequency of this clock must be less than or equal to half the frequency of CPMC440CLK.
CPMINTERCONNECTCLK	Input	1		Main Embedded Processor Block clock for the processor block interconnect (crossbar). This clock is used for the processor PLB interfaces and the processor interconnect (crossbar).
CPMINTERCONNECTCLKEN	Input	1	Async	Clock enable for the embedded processor block interconnect and the processor interfaces. A value of `1` specifies an active clock to the PLB interface. A value of `0` disables the clock. Disabling the clock reduces dynamic power consumption.
CPMINTERCONNECTCLKNTO1	Input	1	Static	Specifies whether the clock ratio between the processor and the interconnect within the embedded processor block is an N:1 integer or a fractional multiple. A value of `1` indicates the clock ratio is an N:1 integer. A value of `0` indicates a fractional clock ratio of (2N+1)/2.
PPCCPMINTERCONNECTBUSY	Output	-	CPMINTERCONNECTCLK	This status signal indicates if any PLB, DMA, or memory controller transactions are active inside the embedded processor block. A `1` indicates transactions are active in the embedded processor block while a `0` indicates no transactions are active. This signal can determine when to reset the embedded processor block or put it in a low power state (disabled clock).
RSTC440RESETCHIP	Input	0	Async	Resets the entire embedded processor block including the processor core.
RSTC440RESETCORE	Input	0	Async	Resets the processor core and the APU.
RSTC440RESETSYSTEM	Input	0	Async	Resets the entire embedded processor block including the processor core.

Clock and Reset During Configuration and Reconfiguration

During FPGA configuration, the embedded processor block in Virtex-5 FPGAs is clocked with an internal FPGA configuration clock and is automatically reset so that it can be ready for operation after configuration. The user need only ensure that the CPMINTERCONNECTCLK clock signal is glitch-free.

During a Static Reconfiguration or Grestore event, the embedded processor block repeats the reset and startup sequence. During an active reconfiguration, the embedded processor block is not automatically reset, but can be reset by the user using the RSTC440RESETCHIP, RSTC440RESETCORE, or RSTC440RESETSYSTEM control signals.

System-Level Considerations

The embedded processor block in Virtex-5 FPGAs internally buffers, conditions, and synchronizes clock, reset, and power management signals for the user to simplify timing and behavior. However, some system level considerations must be made.

The RSTC440RESETCHIP, RSTC440RESETCORE, and RSTC440RESETSYSTEM control signals can be asserted asynchronously to the embedded processor block. The user only needs to ensure that the reset pulse width is sufficient to be detected by the embedded processor block. The CPMINTERCONNECTCLK signal must be stable and running before the reset signals are deasserted. The embedded processor block internally resynchronizes the reset signals, and holds them for the proper number of clock cycles. It takes eight CPMINTERCONNECTCLK clock cycles for the reset signal to propagate through the entire embedded processor block. During this delay, some of the interfaces will continue to be active for up to eight CPMINTERCONNECTCLK clock cycles after the reset signal is detected.

The processor core records the specific reset signal (one of three signals possible) that was last used to reset the processor core. This record is stored in the DBSR register. However, the reset signals provided by the user are first synchronized and extended for the required number of clock cycles by additional logic in the embedded processor block. This logic always releases the core reset signal last, regardless of how the user reset inputs are sequenced. As a result, the processor core always records the core reset signal as the last reset.

The CPMC440TIMERCLOCK signal is used when the CCR1[TCS] bit is set to 1. This signal can be an asynchronous clock signal because it is internally synchronized to the processor clock. The frequency of this clock must not be greater than half the frequency of CPMC440CLK to ensure that CPMC440TIMERCLOCK can be sampled properly.

Clocking inside the processor block can be disabled to place it in sleep mode to reduce power consumption. The CPMC440CLKEN and CPMINTERCONNECTCLKEN signals can be used to enable or disable the clocks in the processor and embedded processor block interconnect, respectively. The PPCCPMINTERCONNECTBUSY signal indicates if there are any active PLB transactions inside the embedded processor block and can be used to ensure conditions are safe to enter sleep mode.

The clock frequency ratio between processor clock and interconnect clock can be N:1 or (2N+1)/2, where N is an integer greater than 0. Example ratios are 1:1, 2:1, 3:1 or 3:2. The CPMINTERCONNECTCLKNTO1 signal must be statically set to one for ratios of N:1, and to zero for ratios of (2N+1)/2. The latency of transactions between the core and interconnect is improved for integer clock ratios.

Because the processor and crossbar clocks are likely to be running at higher speeds than the fabric clocks, it is highly recommended that the C440RSTxxxRESETREQ signals be synchronized to the clock domain in which they will be consumed.

Clock Insertion Delays and PLL Usage

All clocks used by the embedded processor block must be positive-edge aligned. The only exceptions are local link clocks that do not need a fixed frequency or phase relationship with the other clocks, and the DCR clock when the DCR is operated in asynchronous mode.

The embedded processor block uses its own clock trees to distribute its internal clocks, CPMC440CLK, and CPMINTERCONNECTCLK. As a result, there is a delay between the clock edges presented at the edge of the embedded processor block and the clock edges at the internal flip-flops. The DESKEW_ADJUST attribute in the Virtex-5 FPGA PLL blocks can be set on each clock output to delay the output by an amount that matches the clock insertion delay within the embedded processor block. Clocks connected to the embedded processor block and to the buses and peripherals connected to the block should be generated from a PLL block with the appropriate setting for the DESKEW_ADJUST attribute:

- **`NONE`** for no delay used for CPMC440CLK and CPMINTERCONNECTCLK.
- **`PPC`** for a delay equal to the clock insertion delay within the embedded processor block used for all other clocks (such as the PLB clocks that are synchronous to CPMINTERCONNECTCLK).

The allowed frequency ratios for these clocks with respect to CPMINTERCONNECTCLK are shown Table 6-2.

Table 6-2: **Interface Clock Frequency Ratios**

Clock Signal	Allowed Frequency Ratios With Respect to CPMINTERCONNECTCLK
CPMMCCLK	2:3, OR 1:N, where N is any integer in the range [1:16]
CPMPPCMPLBCLK	1:N, where N is any integer in the range [1:16]
CPMPPCS0PLBCLK	1:N, where N is any integer in the range [1:16]
CPMPPCS1PLBCLK	1:N, where N is any integer in the range [1:16]
CPMDCRCLK	1:N, where N is any integer in the range [1:16]

Table 6-3 shows the relationship between all the clock ratios in the embedded processor block. The listed clock frequencies are examples only. The actual frequency depends on the speed grade of the chip and the design. To determine the maximum frequency, refer to DS202, *Virtex-5 FPGA Data Sheet: DC and Switching Characteristics*.

When operating at 1:1 ratios, the maximum speed specified in the data sheet of each associated IP must not be exceeded.

Table 6-3: **Clock Frequency Examples**

Clock Name	Description	Example Frequency (MHz)	Supported Ratio	Reference Clock
CPMC440CLK	CPU Clock	400	-	-
CPMINTERCONNECTCLK	Crossbar Clock	200	2:3 or 1:N, where N is any integer in the range [1:16]	CPMC440CLK
CPMPPCS0PLBCLK	Slave 0 PLB Clock	100	1/N, where N = 1:16	CPMINTERCONNECTCLK
CPMPPCS1PLBCLK	Slave 1 PLB Clock	100	1/N, where N = 1:16	CPMINTERCONNECTCLK
CPMPPCMPLBCLK	Master PLB Clock	100	1/N, where N = 1:16	CPMINTERCONNECTCLK
CPMMCCLK	Memory Interface Clock	200	2:3 or 1:N, where N is any integer in the range [1:16]	CPMINTERCONNECTCLK
CPMFCMCLK	FCM Clock	100	1/N, where N = 1:16	CPMC440CLK
CPMDCRCLK	DCR Clock	100	1/N, where N = 1:16. Asynchronous	CPMINTERCONNECTCLK
CPMDMA0LLCLK	DMA0 LocalLink Clock	150	Asynchronous	-
CPMDMA1LLCLK	DMA1 LocalLink Clock	150	Asynchronous	-
CPMDMA2LLCLK	DMA2 LocalLink Clock	150	Asynchronous	-
CPMDMA3LLCLK	DMA3 LocalLink Clock	150	Asynchronous	-
JTGC440TCK	JTAG Clock	150	Asynchronous	-
CPMC440TIMERCLOCK	Timer Clock	200	Can toggle by at most one-half of the frequency of the reference clock	CPMC440CLK

Chapter 7

Device Control Register Bus

Introduction

The embedded processor block in Virtex-5 FPGAs, which is a CoreConnect based system-on-a-chip, uses the Device Control Register (DCR) bus for device configuration, and control and status accesses. This chapter provides an overview of the DCR arrangement used in the processor block. Refer to the *CoreConnect Bus Architecture Product Brief* [Ref 2] and *Device Control Register Bus 3.5 Architecture Specifications* [Ref 3] for more information.

Figure 7-1 shows a block diagram of the connections of various DCR blocks within the embedded processor block. This chapter focuses on the DCR controller. Information pertaining to individual DCR masters and slaves can be found in the design specifications associated with the blocks.

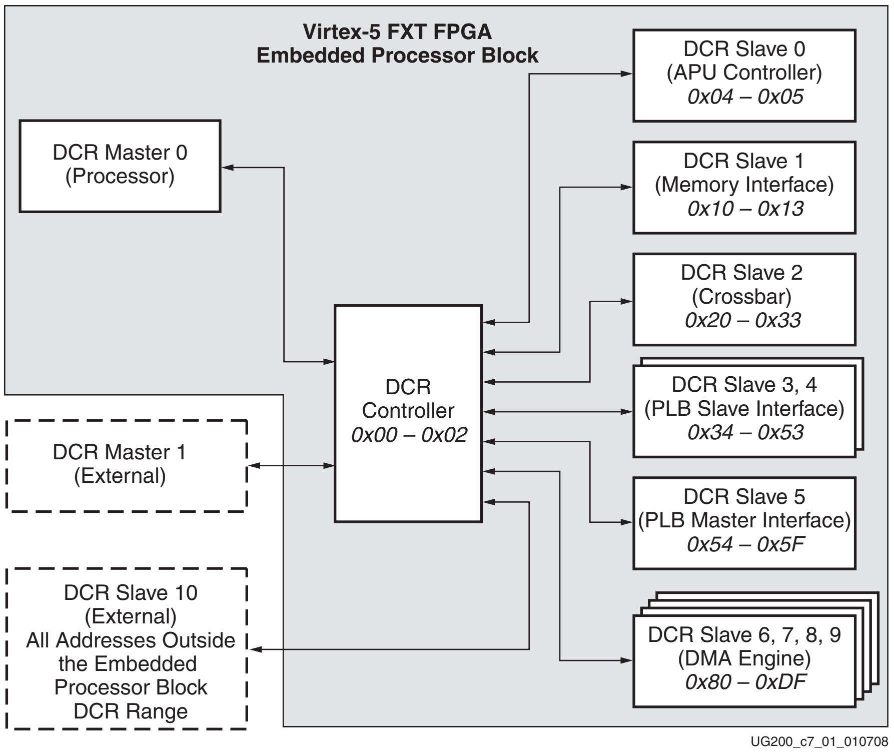


Figure 7-1: **Embedded Processor Block DCR Arrangement (with Address Offsets)**

Figure 7-1 shows the logical connectivity between the controller and the masters and slaves. There are 2 DCR masters and 11 DCR slaves. Of these, one master and one slave are external to the embedded processor block.

A mixed daisy-chain and distributed-OR DCR bus topology (see [Ref 3]) is used within the embedded processor block. In the daisy-chain scheme, data buses are daisy-chained together. In the distributed-OR scheme, the slave data outputs are logically ORed together to form the final output sent to the DCR master. All slave transfer acknowledge signals are ORed together and sent to the DCR master.

The main features of the DCR controller in the embedded processor block in Virtex-5 FXT FPGAs are:

- A mixed daisy-chain and distributed-OR DCR bus topology
- Support for dual DCR masters with round-robin arbitration
- Integrated bus lock capability to allow atomic DCR operation
- Support for indirect addressing
- 10-bit DCR address in direct addressing mode
- 12-bit DCR address in indirect addressing mode
- 32-bit data bus
- Support for the time-out wait feature
- DCR access time-out detection
- Selectable synchronous or asynchronous interface with external DCR devices

Design and Implementation

The DCR controller is situated between the masters and slaves. Commands from the masters are sent through the DCR controller to the DCR slaves, and responses from the slaves are sent to the masters via the DCR controller. The tasks for which the controller is responsible are described in the following subsections.

Partial Address Decoding

The DCR controller carries out partial DCR address decoding to determine to which DCR slave a DCR read/write command is intended. Table 7-1 shows the DCR address map.

The DCR controller supports both direct and indirect addressing modes. In the direct addressing mode, which has a 10-bit address space of 1024 locations, the DCR controller occupies 256 address locations with a starting address that can be configured by two-bit tie-off pins, **TIEDCRBASEADDR**[0:1], to `0x000`, `0x100`, `0x200`, or `0x300`. In the indirect addressing mode (see "Indirect Addressing," page 155), which has an expanded 12-bit DCR address space of 4096 locations, the DCR address space is identical to that in the direct addressing mode and is always located inside the first 1024 locations.

Table 7-1: **DCR Map with Address Offsets**

Block	Address Offset and Range
Indirect Mode Address Register	`0x00`
Indirect Mode Access Register	`0x01`
DCR Controller Status and Control Register	`0x02`
Reserved	`0x03`
Auxiliary Processor Unit (APU) Controller	`0x04 – 0x05`
Reserved	`0x06 – 0x0F`
Memory Interface	`0x10 – 0x12`
Reserved	`0x13 – 0x1F`
Crossbar	`0x20 – 0x33`
PLB Slave 0 (PLBS0)	`0x34 – 0x43`
PLB Slave 1 (PLBS1)	`0x44 – 0x53`
PLB Master (PLBM)	`0x54 – 0x5F`
Reserved	`0x60 – 0x7F`
DMA Engine 0 (DMAC0)	`0x80 – 0x90`
Reserved	`0x91 – 0x97`
DMA Engine 1 (DMAC1)	`0x98 – 0xA8`
Reserved	`0xA9 – 0xAF`
DMA Engine 2 (DMAC2)	`0xB0 – 0xC0`
Reserved	`0xC1 – 0xC7`
DMA Engine 3 (DMAC3)	`0xC8 – 0xD8`
Reserved	`0xD9 – 0xDF`
Reserved	`0xE0 – 0xFF`

Indirect Addressing

The indirect addressing mode allows the DCR slave address to be defined by the content of DCR `0x00` rather than over the DCR address bus. Indirect addressing is carried out through two dedicated DCR locations at offsets `0x00` and `0x01` (as shown in Table 7-1). Both locations are accessible through direct addressing only, and they are the only two DCR locations of the entire DCR address space that cannot be accessed through indirect addressing. Indirect read or write access to either of the two locations results in a DCR time-out (bit 31 of `0x02` is set).

The steps to do indirect addressing are:

1. Through direct addressing, write a 12-bit target DCR address to `0x00`. If configured (see "Register 0x02: Control, Configuration, and Status Register," page 163), this step triggers an auto bus lock action to reserve the bus for indirect access.
2. Through direct addressing, read or write to offset `0x01` as if reading or writing to the target DCR address. This step releases the auto bus lock, if any.

The second step does not need to occur right after the first step. When the target address is written to `0x00`, it stays there until overwritten by another value. The 12-bit DCR address consists of a 2-bit DCR UABUS address and a 10-bit DCR ABUS address (see "DCR Controller Registers," page 162). All 256 DCRs in the embedded processor block in Virtex-5 FPGAs are located within the first 1024 locations for both direct and indirect addressing. Figure 7-2 shows an example of direct and indirect addressing. Register `0x00` is assumed to contain `12'h059` at the beginning. If the ABUS address is `0x01`, address `12'h059` is used (indirect addressing); otherwise the 10-bit ABUS address, preceded by a 2-bit value of `00` is used instead (direct addressing).

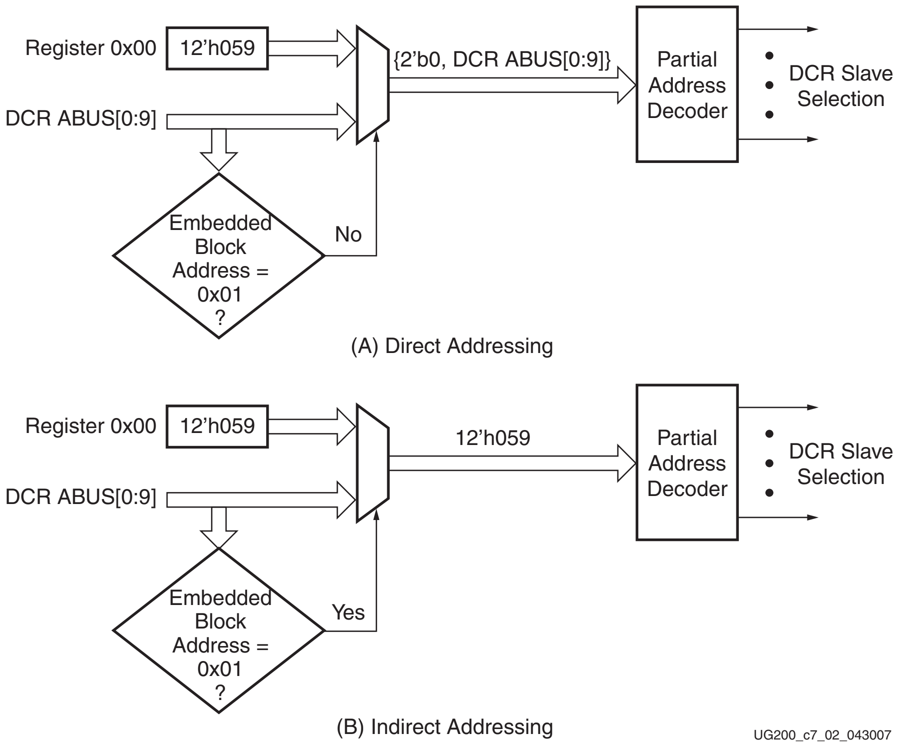


Figure 7-2: **Direct and Indirect Addressing Examples**

Dual DCR Master Arbitration

If either the processor's DCR master or the external DCR master is active, the active master has all the available DCR bandwidth. If both masters are active, only one master can gain access to the DCR slave at a time. Arbitration is based on a Work Conserving Round Robin (WCRR) strategy, where a master who has just accessed a DCR slave has a lower priority for another access unless the other master does not have a pending slave access request. With this arbitration scheme, each master can receive around 50% or more of the available DCR bandwidth if the bus is not locked (see "Bus Lock") by either master. Figure 7-3 shows a simplified arbitration diagram.

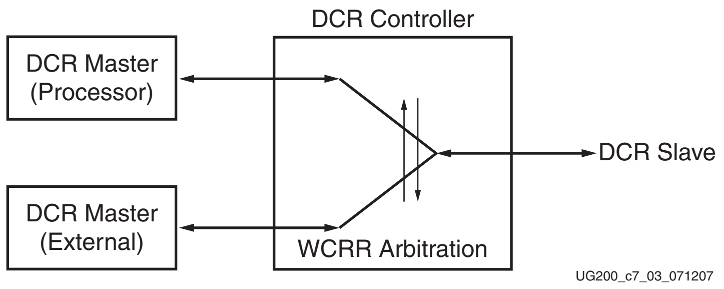


Figure 7-3: **Dual DCR Master Arbitration**

Bus Lock

A consequence of having two active DCR masters is that both masters might attempt to access the same DCR location right after one another, creating a data incoherence situation. The DCR controller provides a bus locking mechanism to help to prevent this situation from happening. When the bus is locked, the locking master has exclusive access to the DCR slaves, and the other master must wait until the locking master releases the lock. It is, therefore, important that the locking master does not hold on to the DCR bus for too long a time to adversely affect the other master. The DCR controller has two types of bus locks: normal and auto.

Normal Bus Lock

The master that wants to lock the bus writes a `1` to a particular bit in register `0x02` (see "Register 0x02: Control, Configuration, and Status Register," page 163). If the processor DCR master wants to lock the bus, it writes a `1` to bit 0 of register `0x02`. If the external master wants to lock the bus, it writes a `1` to bit 2. If the bus is already locked by a master, a lock request by the other master might not be successful, depending on whether the lock requesting master supports time-out waits[1] (see "Time-out Wait," page 159 for more details) and how long the first master holds the lock.

If the lock-requesting master supports time-out waits, it acquires the lock as soon as the locking master releases the lock. However, if the lock-requesting master does not support time-out waits, and the locking master holds the lock for a sufficiently long time, a DCR time-out for the lock-requesting master might occur, resulting in an unsuccessful lock (the lock bit is not set in this case). It is important for an external master that does not support time-out waits to read back register `0x02` to confirm a successful lock.

Auto Bus Lock

When a master writes to register `0x00` to update an indirect address, the DCR bus, by default, is automatically locked for that master until it reads from or writes to `0x01` to release the lock. This auto-lock feature, which can be disabled by clearing bit 4 of register `0x02`, reduces the number of DCR operations required for an atomic indirect access operation. It is important that the master accesses register `0x01` after writing to `0x00` to release the lock, otherwise the bus remains locked and the other master cannot access the bus. The auto bus lock status can be read from register `0x02`. Bit 1, if set, indicates an auto bus lock is active for the processor, and bit 3 serves the same function for the external master. Bits 1 and 3 are read only.

1. When there is a master that does not support time-out waits with another master that might lock the bus, the first master must lock the bus for reliable operation, unless it is certain that the second master will not lock the bus for an extended period of time, leading to a time-out in the first master.

Because only one master can lock the bus at a time, for register `0x02`, bit 0, bit 1, or both can be set, or bit 2, bit 3, or both can be set. Both bits 0/1 and bits 2/3 cannot be set.

While the bus is locked by a master, the other master can read but cannot modify the content of register `0x02` (see Figure 7-4). Both DCR accesses (by the locking master and the read of register `0x02` by the other master) share the same available bus bandwidth.

Round-Robin Arbitration

The round-robin arbitration with the bus lock feature is based on a four-state finite state machine as shown in Figure 7-4.

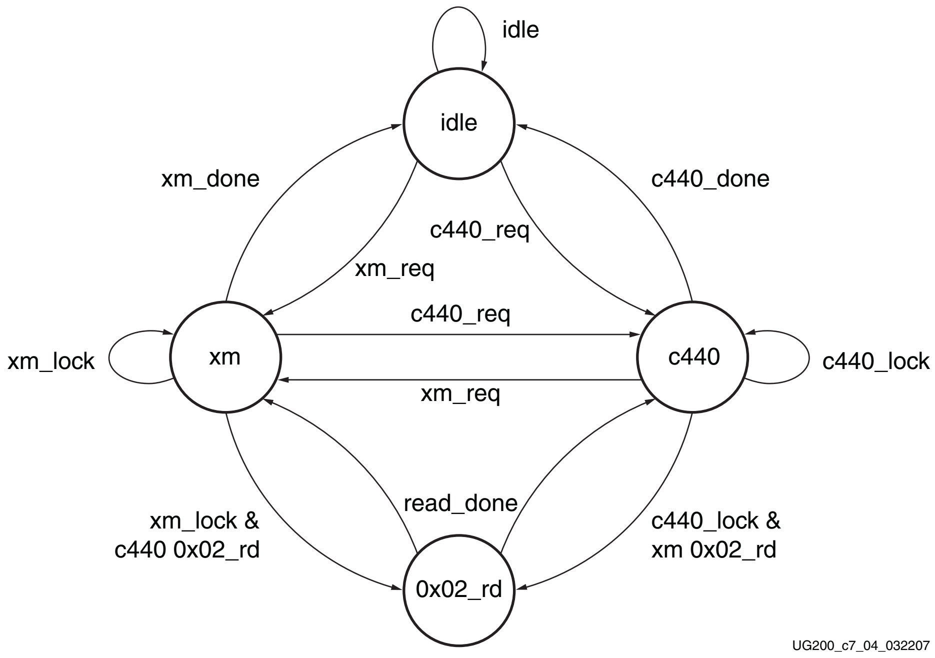


Figure 7-4: **A 4-State Arbitration State Diagram with Simplified Branches**

The arbiter operates in the following manner:

- The arbiter is in the idle state when there is no DCR access.
- The arbiter goes into the c440 state if the current state is idle AND there is a processor DCR request that is either the only request OR the preferred request (if the external master is also requesting)
- When the processor access is done, the arbiter does one of the following:
    - Goes to the idle state
    - Remains in the c440 state if there is a bus lock condition,
    - Goes to the xm state if there is an external master request, or
    - Goes to the 0x02_rd state if there is a bus lock condition AND the external master wants to read register `0x02`.
- If the arbiter is in the 0x02_rd state, it returns to the locking master state after the DCR access.

Similar state transitions occur for the external master, with xm replacing c440, and vice-versa, in the above bullets.

Time-out Wait

The time-out wait signal in a DCR bus is used to inhibit the time-out counter in a master from counting. The DCR controller propagates the time-out wait signal from the external DCR slave to one of the two DCR masters. The embedded processor block's DCR slaves do not generate time-out wait signals, and so no propagation is necessary.

When the DCR bus is locked by a master, the time-out wait signal to the other master is set to temporarily inhibit the time-out wait count in that master. This restraint prevents the bus lock period from being counted as part of the wait time for the other master. It is possible that the external DCR master does not support time-out waits. Figure 7-5 shows the time-out wait arrangement in the DCR controller.

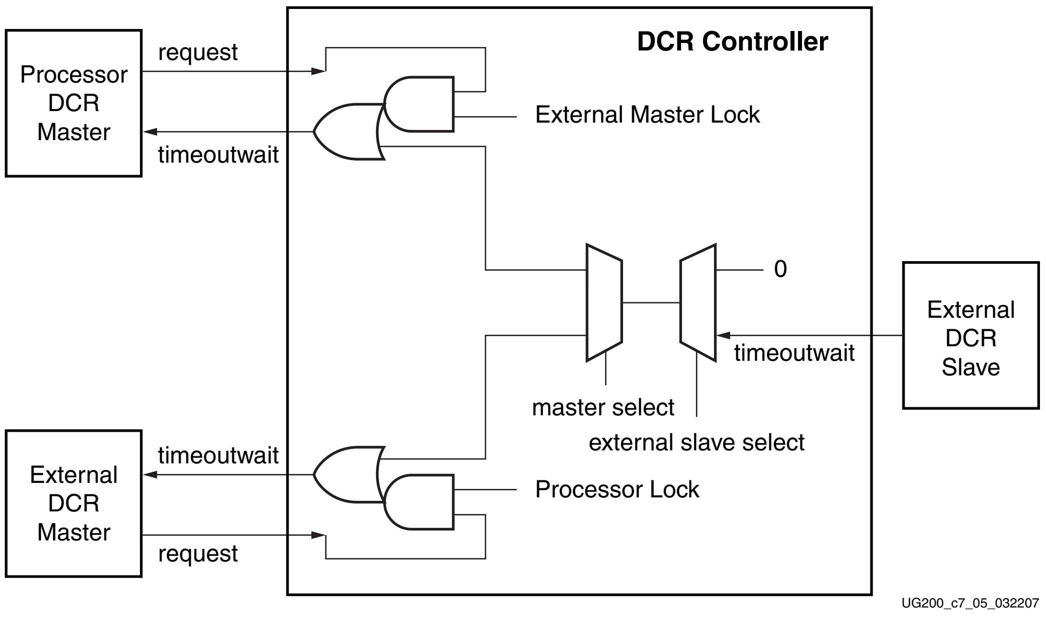


Figure 7-5: **Time-out Wait Arrangement**

Input and Output Interfaces

The interface signals with the external DCR master and slave are shown in Table 7-2 and Table 7-3.

Table 7-2: **DCR Controller's Slave Port Signals (Connected to the External Master)**

Signal Name	Direction	Description
DCRPPCDSREAD	I	Master DCR read command
DCRPPDCSWRITE	I	Master DCR write command
DCRPPCDSABUS[0:9]	I	Master DCR address bus
DCRPPCDSDBUSOUT[0:31]	I	Master DCR data bus out
PPCDSDCRACK	O	Slave DCR acknowledge
PPCDSDCRDBUSIN[0:31]	O	Slave DCR bus in (for the master)
PPCDSDCRTIMEOUTWAIT	O	Slave DCR time-out wait indicator

Table 7-3: **DCR Controller's Master Port Signals (Connected to the External Slave)**

Signal Name	Direction	Description
PPCDMDCRREAD	O	Master DCR read command
PPCDMDCRWRITE	O	Master DCR write command
PPCDMDCRUABUS[20:21]	O	Master DCR upper address bus
PPCDMDCRABUS[0:9]	O	Master DCR address bus
PPCDMDCRDBUSOUT[0:31]	O	Master DCR data bus out, 0 when idle
DCRPPCDMACK	I	Slave DCR acknowledge
DCRPPCDMDBUSIN[0:31]	I	Slave DCR bus in (for the master)
DCRPPCDMTIMEOUTWAIT	I	Slave DCR time-out wait indicator

Table 7-4 defines the input clocks for the DCR controller.

Table 7-4: **DCR Controller's Input Clocks**

Signal Name	Direction	Description
CPMINTERCONNECTCLK	I	Embedded processor block interconnect clock
CPMDCRCLK	I	DCR clock, optional, needed for synchronous interface only

Interface Timings

The DCR bus is a positive-edge synchronous bus. The controller supports both synchronous as well as asynchronous external DCR devices.

In the synchronous interface mode, the external DCR clock frequency has to be an integer fraction (between 1 and 1/16) of the interconnection clock frequency (**`CPMINTERCONNECTCLK`**), and the DCR clock has to be edge-synchronous to that clock. Clock insertion delays must be taken into account when using a PLL to generate the DCR clock, and the DCR clock output of the PLL must have its DESKEW_ADJUST attribute set to **PPC**.

In the asynchronous interface mode, the DCR clock does not have to be synchronous to any clock ratio. The asynchronous interface approach results in an increase in latency and a drop in throughput.

The master and slave interfaces can be configured independently to operate either synchronously or asynchronously. The interface modes after reset are defined by two attribute bits that are defined in this way:

- **PPCDS_ASYNCMODE** (external master interface):
  - 0: Synchronous mode
  - 1: Asynchronous mode
- **PPCDM_ASYNCMODE** (external slave interface):
  - 0: Synchronous mode
  - 1: Asynchronous mode

If either or both interfaces are to operate synchronously, an edge-synchronous DCR clock must be applied to the embedded processor block. If both master and slave operate asynchronously, no DCR clock is required (the unused DCR clock pin has to be tied to 1). There is only one DCR clock input to the embedded processor block. This clock input supports the use of either a synchronous DCR master or a synchronous DCR slave, or both synchronous DCR master and slave, if they run at the same frequency. If the master and slave run at different frequencies, only one can run in the synchronous mode, and the other has to run in the asynchronous mode. The DCR clock input must correspond exactly to the DCR clock used by the synchronous DCR master or slave; that is, it must not be either a multiple of the DCR clock or divided down from the DCR clock.

Asynchronous Mode

In asynchronous mode, synchronization of the interface signals is needed where the read and write command signals from a master are synchronized (for example, through double flip-flopping) by the slave who receives them, as shown in Figure 7-6. The acknowledge and the time-out wait signals from the slave are synchronized by the master who receives them. The address bus and the data bus signals are synchronized through the DCR protocol. The signals from a sender should be glitch-free.

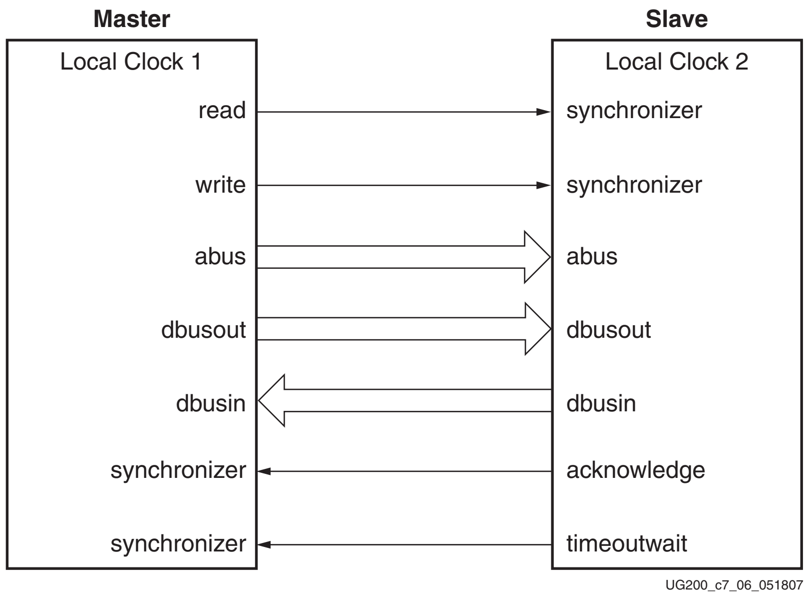


Figure 7-6: **Synchronization for the Asynchronous Interface Mode**

Interface Timing Diagram

The general timing requirements follow that of the DCR bus architecture specifications. Figure 7-7 shows a typical timing diagram (clocks not shown), which is applicable to both synchronous and asynchronous modes (see [Ref 3] for more information).

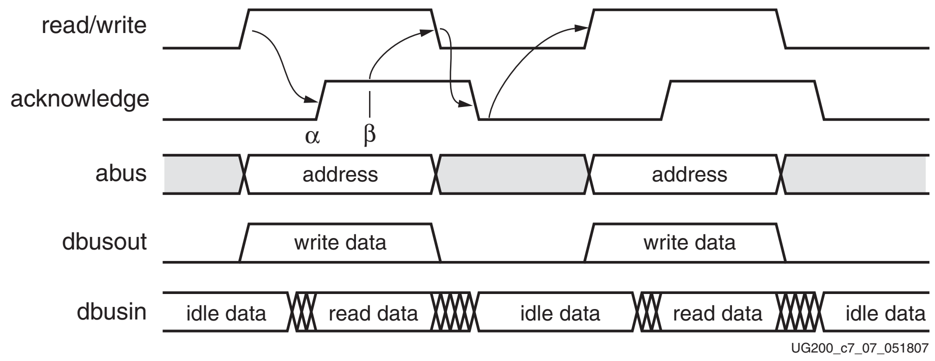


Figure 7-7: **DCR Timing Diagram**

In Figure 7-7, the slave asserts the acknowledge signal when a write operation is complete or when read data has been placed on the bus. As a result the slave should sample the write data at point α, and the master should sample the read data at point β. The dbusin buses from internal slaves are driven to 0 when not active. The DCR controller drives bypass data (dbusin is the same as dbusout) onto the dbusin bus for both the external and internal masters when not active.

DCR Controller Registers

There are three registers in the DCR controller. These registers are needed for indirect addressing, arbitration, and interface mode select.

Register 0x00: Indirect Address Register

This register contains the address used in indirect addressing. The indirect address is formed by a 2-bit upper address bus (UABUS[20:21]) value and a 10-bit address bus (ABUS[0:9]) value. This register, shown in Figure 7-8, is both readable and writable. All unused bits in the register return 0s when read.

0–19	20–21	22–31
Reserved	UABUS[20:21]	ABUS[0:9]

Figure 7-8: **Register 0x00**

Register 0x01: Indirect Access Register

This location is used as a proxy to indirectly access the DCR slaves. When location `0x01` is accessed, the DCR controller replaces the DCR address (`0x01`) with the content of register `0x00` for address decoding. The DCR master reads or writes to the 12-bit address stored in register `0x00`. This location is both readable and writable.

Register 0x02: Control, Configuration, and Status Register

Register `0x02`, shown in Figure 7-9, handles control, configuration, and status. Table 7-5 describes the fields within the register.

0	1	2	3	4	5	6	7	8 … 29	30	31
c440 lock	c440 alock	xm lock	xm alock	auto-lock	xm asyn	xs asyn	xm towait	Reserved	c440 time out	xm time out

Figure 7-9: **Register 0x02**

Table 7-5: **Bit Descriptions for Register 0x02**

Bit	Name	Dir	Default Value	Description
0	c440 lock	R/W	0	Processor bus lock bit. Can be written to and read by the processor DCR master. The external master can also read this bit.
1	c440 alock	RO	0	Processor auto bus lock bit.
2	xm lock	R/W	0	External master bus lock bit. Can be written to and read by the external DCR master. The processor DCR master can also read this bit.
3	xm alock	RO	0	External DCR master auto bus lock bit.
4	auto-lock	R/W	1	Configures the auto-lock feature. The default value for this bit is 1 to enable the auto-lock. This bit is cleared to disable the auto-lock function. This bit is initialized by the embedded processor block attribute DCR_AUTOLOCK_ENABLE.
5	xm asyn	RO	0	Indicates the external DCR master interface asynchronous mode. • 0: Synchronous mode • 1: Asynchronous mode This bit is initialized by the embedded processor block attribute PPCDM_ASYNCMODE.
6	xs asyn	RO	0	Indicates the external DCR slave interface asynchronous mode. • 0: Synchronous mode • 1: Asynchronous mode This bit is initialized by the embedded processor block attribute PPCDS_ASYNCMODE.
7	xm towait	R/W	0	Configures the external DCR master time-out wait support. By default, this bit is 0, so that the external DCR master is assumed not to support time-out waits (the signal is tied to 0), but this setting also works with a master that supports time-out waits. This bit is set to 1 if the external master supports time-out waits, allowing for better performance for the external master if the processor DCR Master locks the bus.
8:29	Reserved	-	0	Reserved.

Table 7-5: **Bit Descriptions for Register 0x02 *(Cont'd)***

Bit	Name	Dir	Default Value	Description
30	c440 timeout	Read/ Clear	0	Set if a processor DCR master access time-out occurs. This bit is cleared on writes. If the bus is locked, only the locking master can clear it, and the other master can read it but not clear it.
31	xm timeout	Read/ Clear	0	Set if an external DCR master access time-out occurs. This bit is cleared on writes. If the bus is locked, only the locking master can clear it, and the other master can read it but not clear it.

The DCR controller prevents more than one master from locking the bus, so writing to bit 0 or 2 might not lead to changes in those bit locations.

Chapter 8

Interrupt Controller Interface

Functional Description

The Interrupt Controller interface allows an external interrupt controller to send interrupts to the processor. An interrupt output signal is generated whenever any of the devices connected to the PLB interfaces of the crossbar raises an error or interrupt signal, as described in Chapter 3, "Crossbar."

Related Processor Behavior

The PowerPC embedded architecture defines two architected interrupts: external and critical. Because critical interrupts have precedence over external interrupts, a critical interrupt can interrupt the processing of an external interrupt. The interface provides dedicated input signals for each of these interrupts. Table 8-1 shows how each of these interrupts is enabled and lists the registers that the processor uses to save the machine state.

Table 8-1: **Architected Interrupts**

Designation	Enabled by	PC and MSR Saved in	Vector Offset from	Description
External Interrupt	MSR[EE]	SRR0, SRR1	IVOR4	When external interrupts are enabled and no critical interrupt is asserted, the processor: • Completes the current instruction (except loads/stores, which might be partially completed) • Saves the next program counter (PC) and the current MSR in the SRR0 and SRR1 registers, which are part of the PowerPC architecture. More information can be found in the *PPC440x5 CPU Core User's Manual* [Ref 5]. • Disables external interrupts • Resumes execution at the PC formed by IVPR and IVOR4. IVPR and IVOR4 are part of the PowerPC embedded architecture. More information can be found in the *PPC440x5 CPU Core User's Manual*, [Ref 5]. Upon executing the **`rfi`** instruction, the processor re-enables external interrupts, loads the PC and the MSR from their saved locations, and resumes execution.
Critical Interrupt	MSR[CE]	CSRR0, CSRR1	IVOR0	When critical interrupts are enabled, the processor: • Completes the current instruction (except loads/stores, which might be partially completed) • Saves the next PC and the current MSR in the CSRR0 and CSRR1 registers, which are identical to the SRR0 and SRR1 registers, except they are in effect for critical interrupts only • Disables all interrupts • Resumes execution at the PC formed by IVPR and IVOR0 Upon executing the **`rfci`** instruction, the processor re-enables interrupts, loads the PC and the MSR from their saved locations, and resumes execution.

On-Core Interrupt Sources

Table 8-2 shows the on-core sources of each type of interrupt.

Table 8-2: **On-Core Interrupt Sources**

Interrupt Type	Source	Description
External Interrupt	FIT	Fixed Interval Timer
	DEC	Decrement Timer
Critical Interrupt	Watchdog timer expiration	Second watchdog timer expiration

Interrupt Interface Signals

The Interrupt Controller interface consists of dedicated inputs for the external and critical interrupts. Both inputs are active High and level sensitive (once asserted, the signal must remain asserted until explicitly cleared by system software).

Each input includes a metastability flop, so its source need not meet any timing requirements (it can be treated as a false path).

Table 8-3 lists the Interrupt Interface signals.

Table 8-3: **Interrupt Controller Interface Signals**

Signal Name	Description
EICC440EXTIRQ	External Interrupt input
EICC440CRITIRQ	Critical Interrupt input
PPCEICINTERCONNECTIRQ	Interrupt request from the crossbar. Users can connect this signal to one of the interrupt input signals or to an interrupt controller that combines interrupt signals from various sources.

Usage Requirements

The requirements in this section must be met to correctly use the Interrupt Controller interface.

The level-sensitive treatment of the inputs by the processor requires that an interrupt signal, once asserted, remains asserted until explicitly cleared by system software. This requirement implies that software-accessible register(s) must be available on the interrupt controller for this purpose. It is critical that the processor's transactions with these registers complete atomically to avoid accidentally handling the same interrupt multiple times. For example, if the write transaction to clear an interrupt was delayed by write posting, the ISR might return before the IRQ line deasserts. This situation would cause the processor to be immediately interrupted again even though the cause is the same interrupt event. The effects of this situation range from reduced performance to data loss or failure, depending upon the application (for example, IRQ handlers for peripherals with clear-on-read registers could lose data). These problems can be avoided by using either the Sync TAttribute described in Chapter 3, "Crossbar," or the DCR interface for these registers.

Chapter 9

JTAG Interface

The JTAG interface, on the embedded processor block in Virtex-5 FXT FPGAs, provides the ability for an external debug tool to gain control of the processor for debug purposes. Through the JTAG interface and using the debug facilities designed into the processor core, a debugger can single step the processor and interrogate internal processor states to facilitate hardware and software debugging.

There are two recommended ways of connecting the JTAG interface on the embedded processor block:

- Use the TAP controller inside the processor independently by connecting the JTAG interface signals directly to the FPGA programmable I/Os.
- Daisy-chain the processor's TAP controller with the FPGA's TAP controller using the JTAG_PPC Processor IP module supplied by Xilinx as part of the software and IP products.

With one exception, the JTAG interface follows IEEE Standard 1149.1, which defines a test access port (TAP) and Boundary-Scan architecture. In the standard, TRST is listed as an optional signal but the JTGC440TRSTNEG signal is required in the embedded processor block. The JTAG interface of the FPGA does not provide this optional TRST pin. The JTGC440TRSTNEG signal must be wired to user I/O or internally tied High. When wiring to user I/O, place an external 10 KΩ pull-up resistor on the trace. Refer to "JTGC440TRSTNEG," page 171 for details. Other than this exception, the JTAG interface supports user-specific instructions, as allowed by the standard, which provide the ability to gain control of the processor for debug.

JTAG Interface I/O Symbol

Figure 9-1 illustrates the inputs and outputs of the JTAG interface.

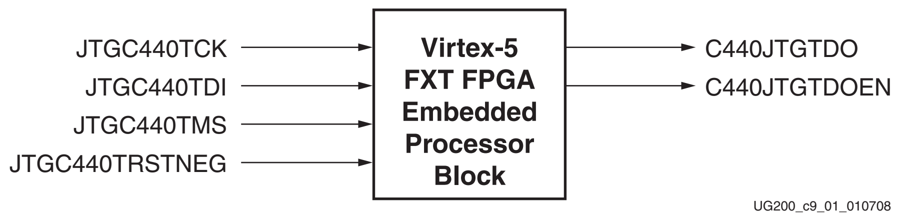


Figure 9-1: **JTAG Interface Block Symbol**

JTAG Interface I/O Signal Descriptions

Table 9-1 describes the JTAG interface signals in alphabetical order.

Table 9-1: **JTAG Interface I/O Signals**

Signal	I/O Type	If Unused	Function
C440JTGTDO	O	No Connect	JTAG Test Data Out (TDO). JTAG serial data out port. This signal transmits data from the processor's TAP. Data from the selected TAP shift register is shifted out on TDO. This serial output from the test logic is fed from either the instruction register or a test data register, depending on the sequence previously applied at TMS. During shifting, data applied at TDI appears at TDO after a number of TCK cycles determined by the length of the register included in the serial path. The signal driven through TDO changes state following the falling edge of TCK. When data is not being shifted through the chip, TDO should be three-stated.
C440JTGTDOEN	O	No Connect	The processor's driver enable signal for the JTAG TDO signal. This signal is the driver-enable signal to the FPGA's three-state driver for the Test Data Out (TDO) signal.
JTGC440TCK	I	See IEEE 1149.1	JTAG Test Clock (TCK). The processor's JTAG logic source clock. TCK is the source clock for the processor's TAP. This clock is independent of the system clock(s) for the chip so that test operations can be synchronized between the various chips on a printed wiring board. Both the rising and falling edges of this clock are significant. The rising edge is used to load signals applied at the TAP input pins (TMS) and (TDI), while the falling edge is used to clock signals out through the TAP TDO pin.
JTGC440TDI	I	High	JTAG Test Data In (TDI). JTAG serial data in port. TDI is used to input serial data into the TAP. When the TAP enables the use of the TDI signal, the TDI signal is sampled on the rising edge of TCK, and this data is input to the selected TAP shift register. Data applied at this serial input is fed into the instruction register or into a test data register, depending on the sequence previously applied at TMS. Typically, the signal applied at TDI is controlled to change state following the falling edge of TCK, while the registers shift in the value received on the rising edge. Like TMS, TDI should be equipped with a pull-up resistor or otherwise designed such that, when it is not driven from an external source, the test logic perceives a logic 1.

Table 9-1: **JTAG Interface I/O Signals *(Cont'd)***

Signal	I/O Type	If Unused	Function
JTGC440TMS	I	High	JTAG Test Mode Select (TMS). Determines the mode in which the TAP operates. TMS is sampled by the TAP on the rising edge of TCK. The TAP state machine uses TMS to determine the mode in which the TAP operates. The operation of the test logic is controlled by the sequence of 1s and 0s applied at this input, with the signal value typically changing on the falling edge of TCK. This signal sequence is fed to the TAP controller, which samples the value at TMS on each rising edge of TCK. The TAP controller uses this information to generate the clock and control signals required by the other test logic blocks. On the chip, TMS should be pulled High when it is not driven from an external source.
JTGC440TRSTNEG	I	High	TRST provides an asynchronous reset of the TAP controller. If this signal is asserted to a logic 0, the TAP controller is asynchronously reset to the Test-Logic-Reset controller state. This signal is negative active at the processor boundary. The designer can connect this signal to a TRST chip input pin. During the power on reset (POR) sequence, the JTGC440TRSTNEG signal is asserted (driven Low) internally. After that, separate control of the processor's JTAG logic reset and non-JTAG logic reset can be accomplished. The system designer must carefully determine how to make the JTAG reset logically responsive to the system and chip resets, depending on the debug requirements and debug tool requirements.

Connecting PPC440 JTAG Logic Directly to Programmable I/O

The simplest way to access the PPC440 JTAG logic is to wire the processor block's JTAG signals directly to programmable I/O. For devices with multiple PPC440 blocks, users may wire each set of PPC440 JTAG signals directly to programmable I/O (Figure 9-2), chain the processors together with programmable interconnect and wire the combined PPC440 JTAG chain to programmable I/O (Figure 9-3), or multiplex a single set of JTAG pins to multiple embedded blocks (Figure 9-4).

Each of these connection styles requires additional I/O and a separate JTAG chain for the PPC440 embedded block(s). The PPC440 embedded blocks must not be placed in the same JTAG chain as the dedicated device JTAG pins because the chain will be broken by the missing PPC440 JTAG logic prior to FPGA configuration.

The $\overline{\text{TRST}}$ signal, which is not implemented on any Xilinx devices, is available on the IBM PPC440 embedded block. This signal may be wired to user I/O or internally tied High. If wired to user I/O, an external 10 ΩW pull-up resistor should be placed on the trace.

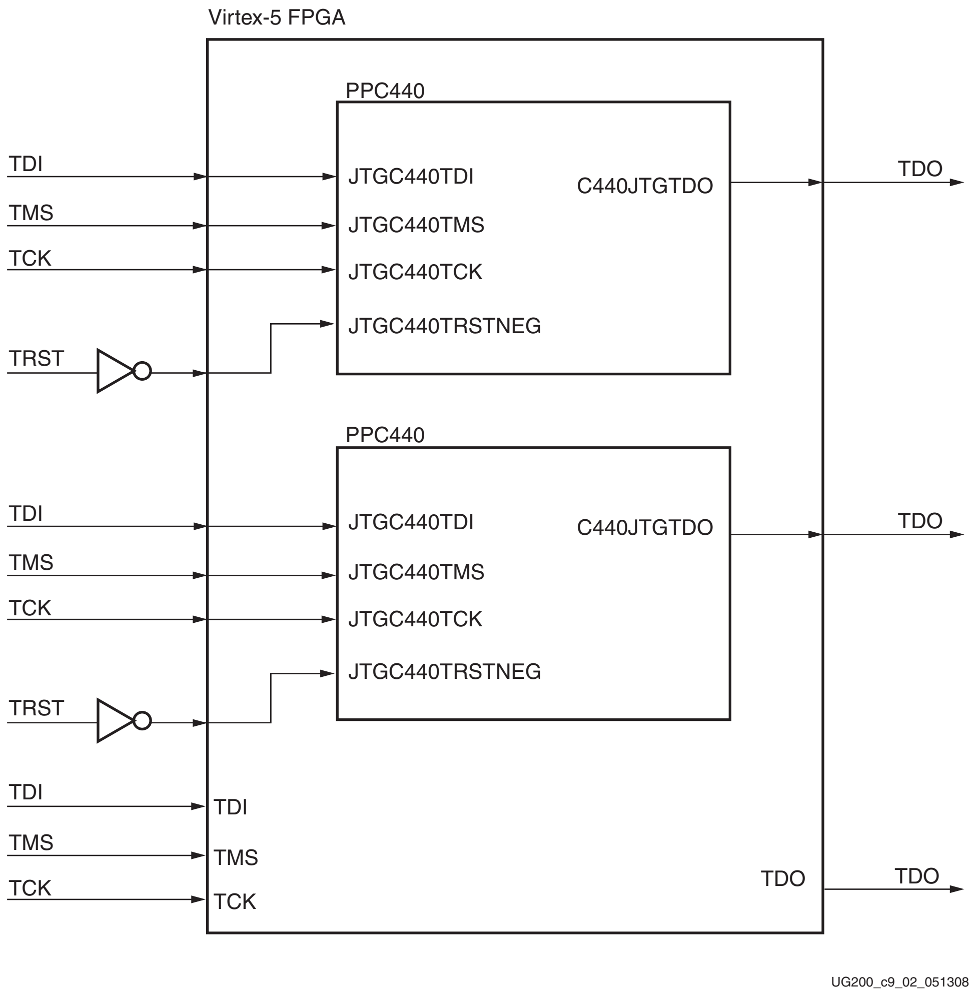


Figure 9-2: **Correct Wiring of JTAG Chains with Individual PPC440 Connections (Separate JTAG Chains)**

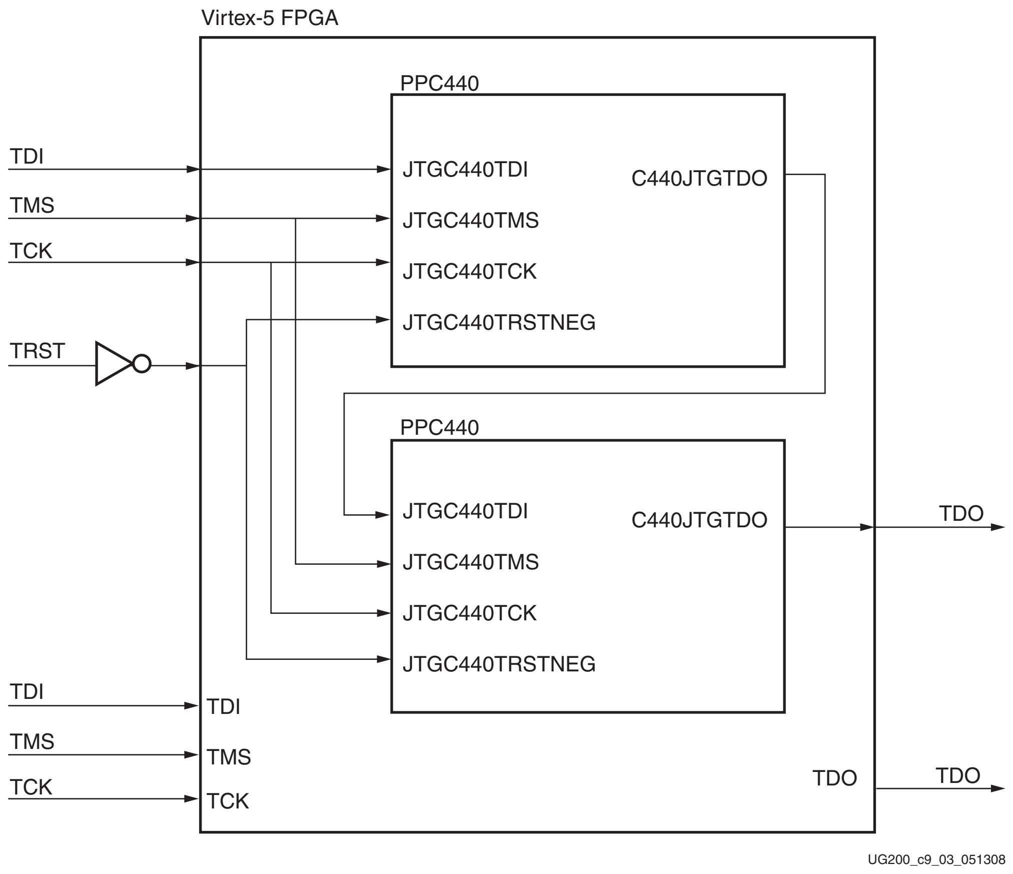


Figure 9-3: **Correct Wiring of JTAG Chains with Individual PPC440 JTAG Connections (Internally Chained PPC440 Embedded Blocks)**

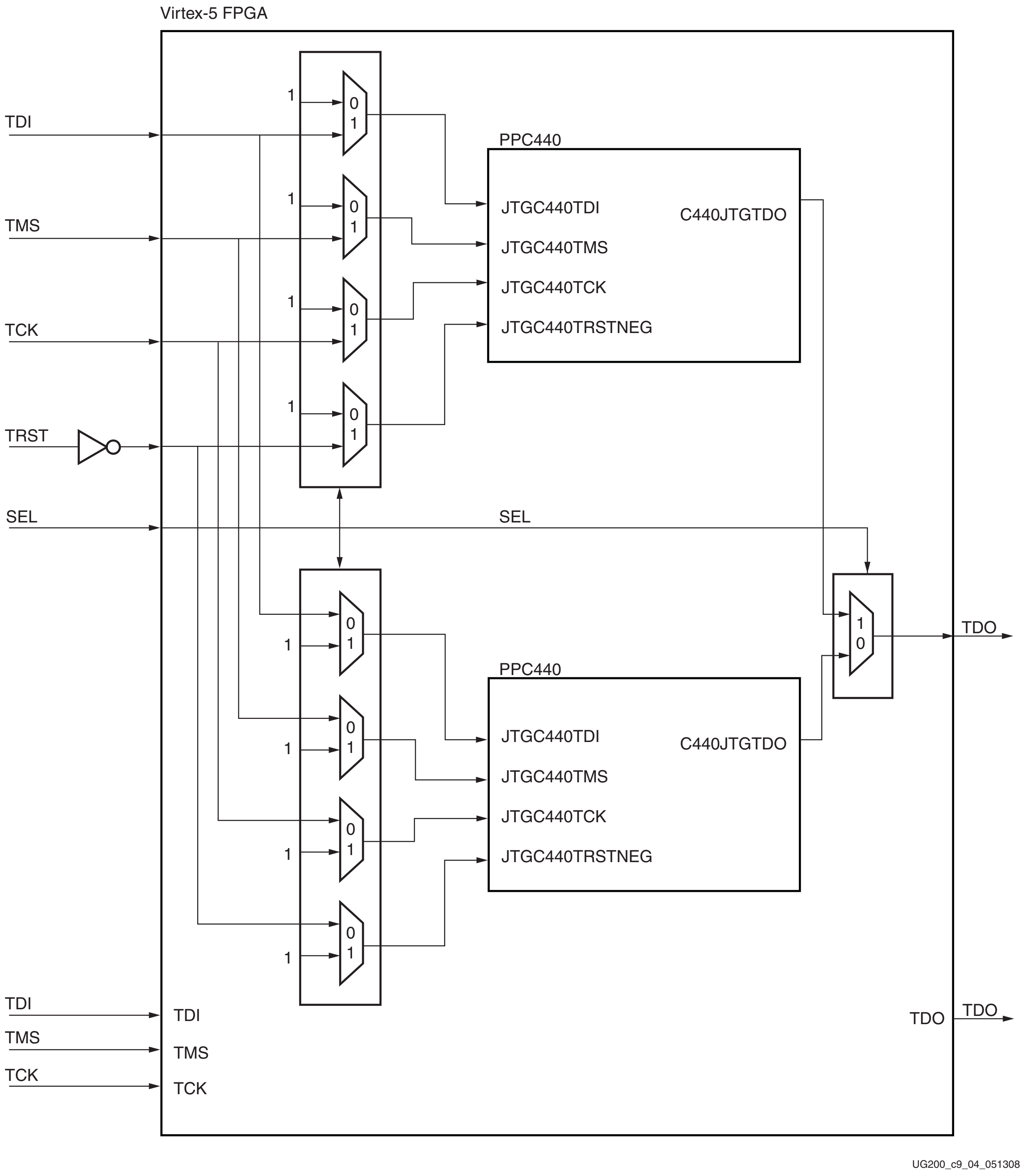


Figure 9-4: **Correct Wiring of JTAG Chain with Multiplexed PPC440 Connection**

Connecting PPC440 JTAG Logic in Series with the Dedicated Device JTAG Logic

An alternative to connecting the PPC440 JTAG logic directly to programmable I/O is to wire it in series with the dedicated device JTAG logic. This is done by wiring the JTAG signals on the PPC440 embedded block to a special design element called the JTAGPPC440 primitive in the user design. The Instruction Register length remains constant, regardless of how the PPC440 embedded blocks are used and regardless of whether or not the device is configured.

Prior to configuration, the most-significant IR bits are placed in a dummy register which is either 4 or 8 bits in length, depending on the number of available PPC440 embedded blocks in the device (4 bits for devices with one PPC440 and 8 bits for devices with two PPC440 blocks). This register is used as a placeholder only. After configuration, if the user connects the PPC440 JTAG logic in series with the dedicated device JTAG logic, the most significant IR bits are used by the PPC440 embedded blocks. Thus, the overall IR length remains the same for the device at all times.

When the PPC440 JTAG logic is connected in series with the dedicated JTAG logic, the C440JTGTDO signal of each embedded block is connected to the JTGC440TDI of the next. The JTGC440TCK and JTGC440TMS signals are connected to each PPC440 embedded block in parallel. The /TRST signal, which is not implemented on the device, is implemented on the IBM PPC440 embedded block. When wiring the PPC440 JTAG logic in series with the FPGA JTAG logic, this signal must be pulled High.

For more information, see the Virtex-5 FPGA user guides.

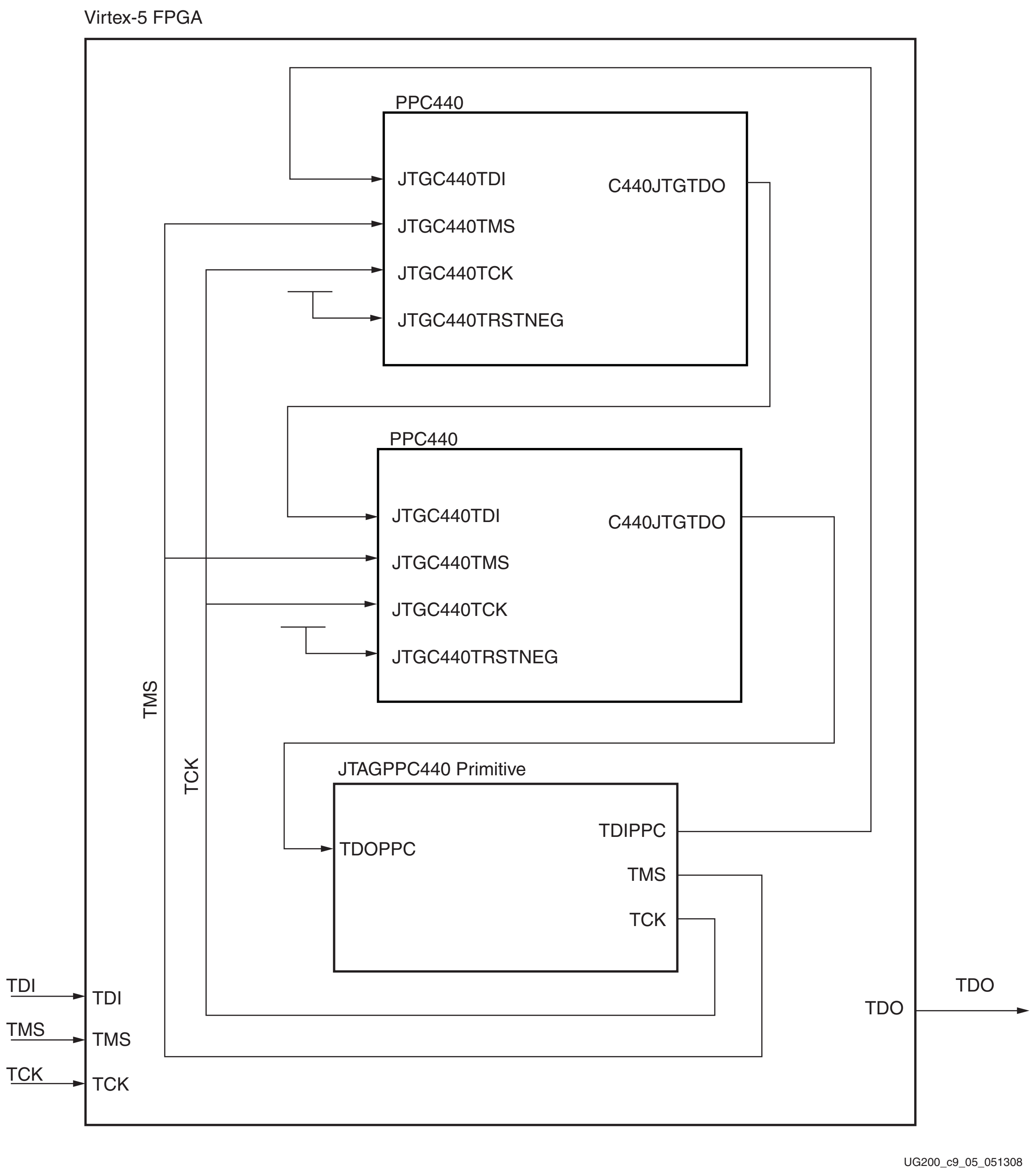


Figure 9-5: **PPC440 Core JTAG Logic Connected in Series with FPGA JTAG Logic Using the JTAGPPC440 Primitive**

When the PPC440 JTAG logic is connected in series with the dedicated device JTAG logic, only one JTAG chain is required on the printed circuit board. All JTAG logic is accessed through the dedicated JTAG pins with this connection style.

For devices with more than one PPC440 embedded block, users must connect the JTAG logic for ALL of the PPC440 embedded blocks on the device when using this connection style, even if some are not otherwise used. The JTAG signals are the only signals on unused PPC440 embedded blocks that need to be connected. The PPC440 embedded block that first sees TDI from the JTAGPPC440 primitive recognizes the first four most significant bits in the Instruction Register; the next PPC440 embedded block sees the next four most significant bits, and so on.

Debug Interface

The Debug interface inputs into the embedded processor block in Virtex-5 FXT FPGAs can provide additional debug enhancements for the designer. The signals on this interface provide information and control to an external debug tool. They also allow customer debug logic to interrupt the normal processor flow through detection and reporting of an off-core debug event.

Debug Interface I/O Symbol

Figure 10-1 illustrates the inputs and outputs of the Debug interface.

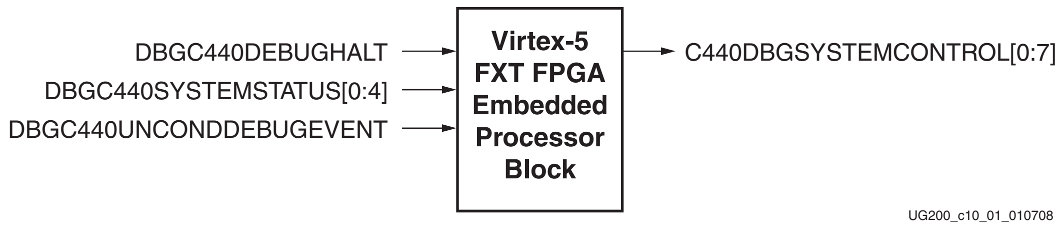


Figure 10-1: **Debug Interface Block Symbol**

Debug Interface I/O Signal Descriptions

Table 10-1 describes the Debug interface signals in alphabetical order.

Table 10-1: **Debug Interface I/O Signals**

Signal	I/O Type	If Unused	Function
C440DBGSYSTEMCONTROL[0:7]	O	No Connect	Reserved by IBM. Leave this signal unconnected.
DBGC440DEBUGHALT	I	0	This signal enables an external source to stop the processor. It connects to a chip pin to allow an external debugger, such as RISCWatch, to request that the processor halt its instruction processing so that the external debugger can control the processor. External debuggers can also issue a stop command to the processor via the JTAG interface. However, this stop request is cleared when the processor is reset, requiring the external debug tool to regain control while the processor is fetching instructions. When using debugHalt to stop the processor, a processor reset does NOT cause the debugHalt control signal to reset, and the processor is stopped at the reset vector. **Note:** The debugHalt chip input on the RISCWatch connector is negative active and needs to be inverted (and synchronized to the processor clock) before being brought into this positive active core input. If the chip has clock control circuitry and the clocks to the processor are turned off (either by external gating or deassertion of CPMC440CLOCKEN, and CPMC440CORECLOCKINACTIVE is driven High), the debugHalt signal should be used by an external debugger as a way to alert clock and power management control logic to re-enable clocks to perform RISCWatch debug activity. If clock control circuitry exists that can prevent the core from getting clocks, and this circuitry can be active during RISCWatch debug activity, the debugHalt signal is required to re-enable clocks to the processor. When the debugHalt signal is deasserted (and no stop request is active on the JTAG interface), the chip should return to the sleep mode it was in before RISCWatch asserts debugHalt, as long as no other condition that would cause the chip to leave sleep mode prevents it from doing so.

Table 10-1: **Debug Interface I/O Signals *(Cont'd)***

Signal	I/O Type	If Unused	Function
DBGC440SYSTEMSTATUS[0:4]	I	0	Reserved by IBM. Connect this signal to 0.
DBGC440UNCONDDEBUGEVENT	I	0	Feeds the UDE bit of the DBSR and allows user debug logic to interrupt normal CPU flow. This input feeds the UDE (unconditional debug event) bit of the DBSR (Debug Status Register). This input is useful for designers who want their own debug logic external to the processor. This capability allows the designer to: • Cause a debug interrupt in internal debug mode • Stop the processor in external debug mode • Send a trigger event code on the processor's trace bus

Trace Interface

The embedded processor block in Virtex-5 FXT FPGAs provides a trace interface that enables the connection of an external trace tool and allows for user-extended trace functions. Users can have full trace capability without adding FPGA logic, although including some trace control logic can provide some benefits.

Trace Interface I/O Symbol

Figure 11-1 illustrates the inputs and outputs of the trace interface.

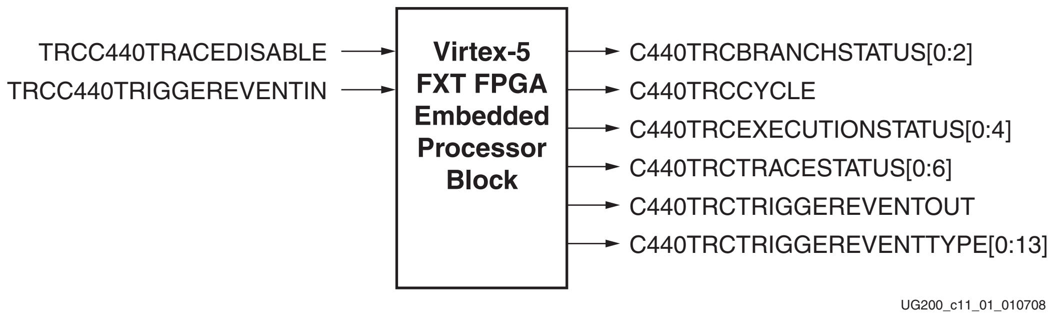


Figure 11-1: **Trace Interface Block Symbol**

Trace Interface I/O Signal Descriptions

Table 11-1 defines the trace interface signals in alphabetical order.

Table 11-1: **Trace Interface I/O Signals**

Signal	I/O Type	If Unused	Function
C440TRCBRANCHSTATUS[0:2]	O	No Connect	Branch status bus for branch instructions represented in the trace cycle (trcCycle). This signal provides branch execution status used by the trace tool in combination with other instruction status to reconstruct the execution flow of a program.
C440TRCCYCLE	O	No Connect	This signal represents the trace cycle. It is used to synchronize the trace period with the four CPU clock cycles. To reduce the amount of chip I/O switching, the core broadcasts new execution status, branch status, and trace status every fourth core cycle. The rising edge of the C440TRCCYCLE signal corresponds with the new trace cycle. This signal is not a clock signal. If it is to be used as a clock, the designer must ensure proper timing of this signal.
C440TRCEXECUTIONSTATUS[0:4]	O	No Connect	Encoded execution status bus for the instructions represented in the trcCycle. To reduce the amount of chip I/O switching, the core broadcasts execution status every fourth core cycle during each broadcast. C440TRCCYCLE defines the cycle that trace data is broadcast.
C440TRCTRACESTATUS[0:6]	O	No Connect	Encoded trace status bus. This signal provides additional information to execution and branch status required by a trace tool to reconstruct the execution flow of a program.
C440TRCTRIGGEREVENTOUT	O	Wrap to Trigger Event In	CPU (debug) trigger event indication for trace logic. Processor-defined debug events can be programmed to create trigger events to external trace logic. This signal is a summary of all processor-defined trigger events. For a trigger event generated by this signal to be in step with execution status for RISCTrace, this signal must be combinatorially returned to the processor on TRCC440TRIGGEREVENTIN within the same clock cycle of the processor clock, CPMC440CLK. In case this timing constraint is hard to meet, a multi-cycle constraint (relaxing the period to 4x the period of CPMC440CLK) can be used. Assuming no special function is required, the designer is required to tie this output to the TRCC440TRIGGEREVENTIN pin. For more information on debug events, refer to the *PPC440x5 Core User Manual*.

Table 11-1: **Trace Interface I/O Signals *(Cont'd)***

Signal	I/O Type	If Unused	Function
C440TRCTRIGGEREVENTTYPE[0:13]	O	No Connect	Identifies the debug event that caused C440TRCTRIGGEREVENTOUT to be asserted. Processor-defined debug events can be programmed to create trigger events to external trace logic. Table 11-2 defines the processor-defined debug events in this bus. (see Table 11-2 below) Designers use these signals to develop more elaborate triggering schemes based on type or sequence of processor-generated trigger events. For a trigger event generated by these signals to be "in step" with execution status for RISC Trace, this signal must be combinatorially returned to the processor on the TRCC440TRIGGEREVENTIN input. For more information on debug events, refer to the *PPC440x5 Core User Manual*.

Table 11-2: **Processor-Defined Debug Events**

C440_trcTriggerEventType Bit	Trigger Event Type
0	Instruction Address Compare 1 (IAC1)
1	Instruction Address Compare 2 (IAC2)
2	Instruction Address Compare 3 (IAC3)
3	Instruction Address Compare 4 (IAC4)
4	Data Address Compare 1 (DAC1RD)—Read
5	Data Address Compare 1 (DAC1WR)—Write
6	Data Address Compare 2 (DAC2RD)—Read
7	Data Address Compare 2 (DAC2WR)—Write
8	Trap Instruction (TRAP)
9	Interrupt (IRPT)
10	Unconditional (UDE)
11	Return (RET)

Table 11-1: **Trace Interface I/O Signals *(Cont'd)***

Signal	I/O Type	If Unused	Function
TRCC440TRACEDISABLE	I	0	This signal is used only in special circumstances to disable trace from outside the embedded processor block. For normal operation, this signal should be tied Low.
TRCC440TRIGGEREVENTIN	I	Wrap to Trigger Event Out	Trigger event input to trace logic. The processor uses this input to generate trigger event codes in the trace status bus, C440_traceStatus[0:3]. A trigger event can be derived from processor trigger events, trigger event types, or from any other external source. If this signal is generated with on-chip combinational logic using processor-generated trigger events, the trigger event code on the trace status bus corresponds to the current execution status being broadcast. The RISCTrace tool can identify which instruction caused the trigger event. If the trigger event is generated from an external source, the trigger event code on the trace status bus corresponds to the execution status of the current instruction. If the trigger event is not generated from an external source, the designer is required to connect the C440TRCTRIGGEREVENTOUT pin to the TRCC440TRIGGEREVENTIN pin.

Section III: Controllers

XILINX®

Chapter 12

Auxiliary Processor Unit Controller

Overview

The native 440 instruction set can be extended with the Auxiliary Processor Unit (APU) controller. Custom instructions are executed by an FPGA fabric coprocessor module (FCM), also referred to as a coprocessor or auxiliary processor. This module enables a much tighter integration between an application-specific function and the processor pipeline than is possible using a bus peripheral.

The APU controller has two purposes:

- Performs clock domain synchronization between the fast processor clock and the slow FCM interface clock
- Decodes certain FCM instructions and notifies the CPU of the CPU resources needed by the instruction (for example, source data from the CPU's general-purpose registers)

A floating-point unit (FPU) is an example of an FCM candidate. For an FCM FPU, the APU controller can decode all PowerPC floating-point instructions. The FCM interface is a Xilinx adaptation of the native APU interface implemented on the IBM processor. The hard core APU controller bridges the processor APU interface and the external FCM interface. This chapter provides detailed information on the FCM interface and its features.

Feature Summary

The key characteristics and features of the APU controller and FCM are listed in this section.

The APU controller:

- Can hold or stall the processor pipeline at various pipe stages
- Supports instructions that do and do not return data to the processor
- Can pipeline up to three instructions at a time, providing for low communication overhead in the instruction issue to the APU controller
- Decodes all FPU instructions in the Book E specification except for the "extended" FPU load/store instructions not supported by the processor
- Contains up to 16 user-defined instruction (UDI) configuration registers

  The 16 UDIs can decode a full primary and extended opcode or can be configured to only decode a shortened version of the extended opcode. This "wildcard" option allows the FCM to use five bits of extended opcode as it chooses.
- Decodes FCM loads and stores with byte, halfword, word, doubleword, and quadword sizes

- Decodes Vector Multimedia Extension (VMX) instructions, which are a subset of FCM loads and stores (these are also known as Altivec instructions)
- Sends FCM decoded information for user-defined, FPU, and load/store instructions to ease the decoding responsibilities of the FCM
- Has separate 128-bit load and store buses
- Has only one instruction in play at a time with the FCM. If the FCM is pipelined, it can execute multiple instructions at a time that do not return data to the processor.
- Sends a signal to the FCM notifying it of a second instruction that will be sent immediately after the current FCM instruction is finished. This will allow the FCM to do some amount of pipelining if desired and can increase instruction throughput.

The FCM:

- Can run at integer multiples (1:1 up to 16:1) of the processor clock period. The FCM cannot run at a speed faster than the processor.

Interface Description

The APU controller is tightly coupled with the processor pipeline. It tracks each FCM instruction through the processor pipeline to know when to expect certain signals. On the other side, it has a simpler control and data interface with the FCM block, having only one instruction in play at any given time. Figure 12-1 shows the control and data flow between the APU controller and the processor. Figure 12-2 shows the data flow between the APU controller and the FCM.

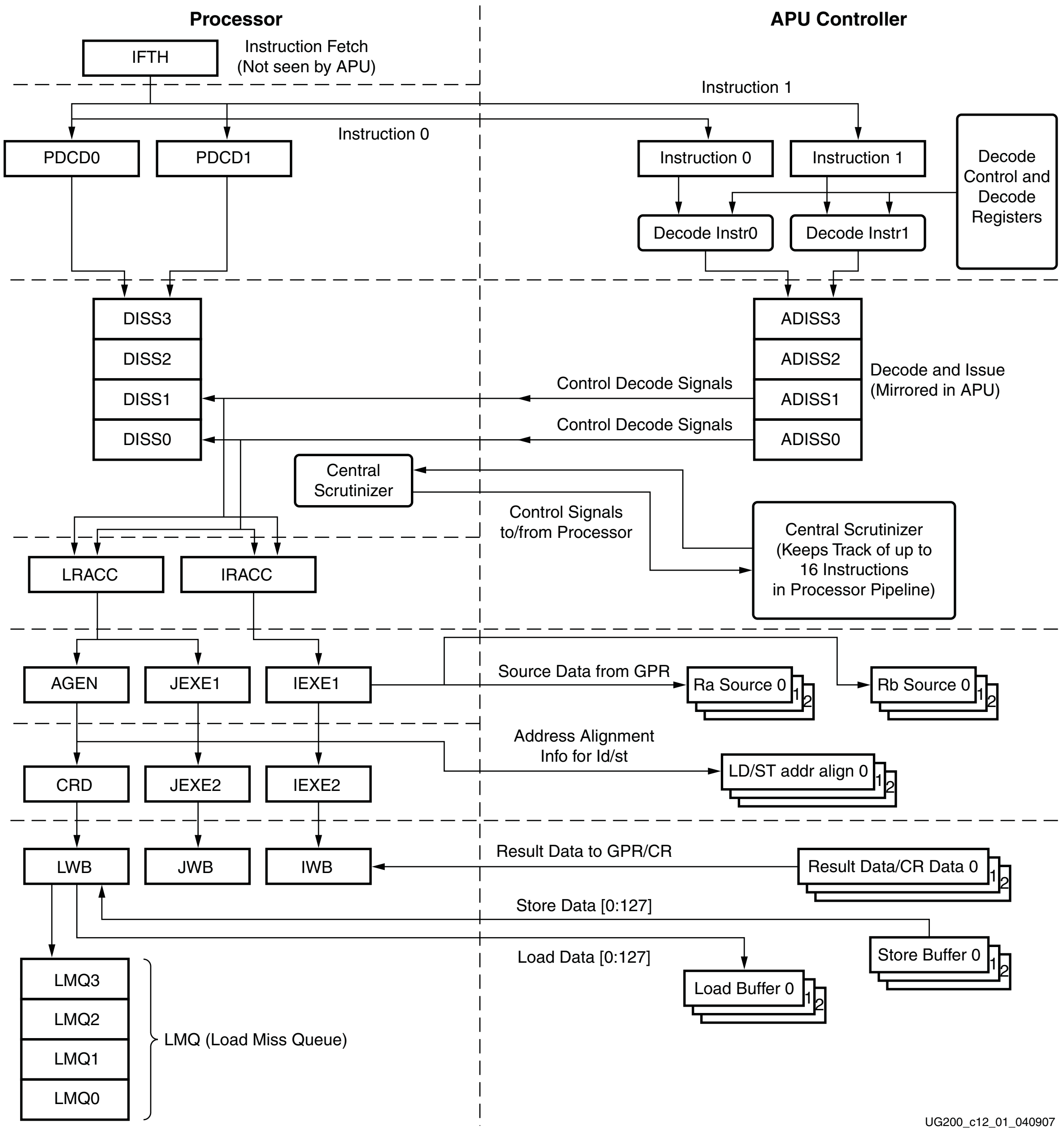


Figure 12-1: **Data Flow between the Processor and the APU Controller**

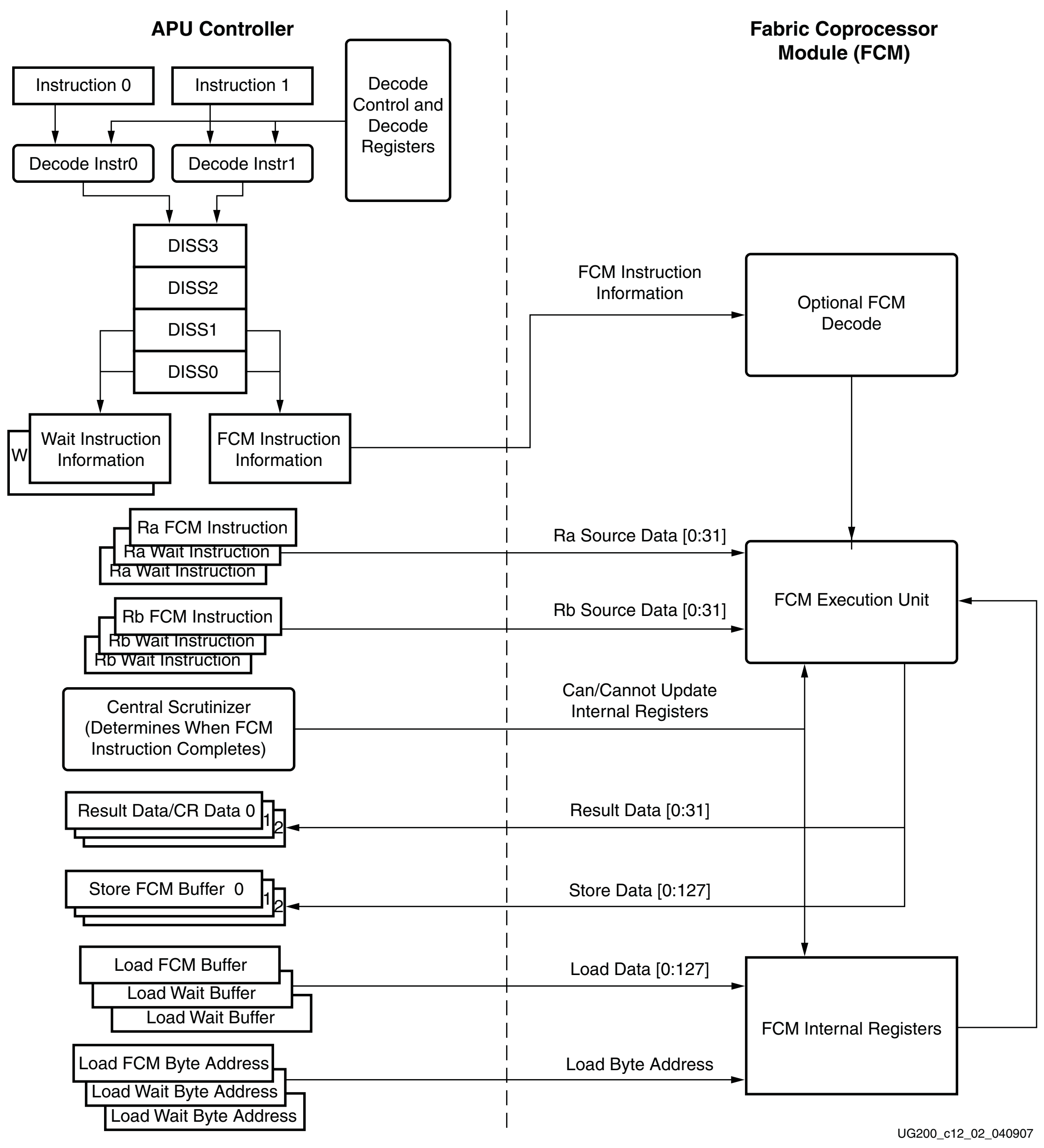


Figure 12-2: **Data Flow between APU Controller and FCM**

Table 12-1 summarizes the signals between the APU controller and the FCM.

Table 12-1: **APU to FCM Signal Descriptions**

Interface Signal	Direction	Function
APUFCMDECFPUOP	Output	APU controller decoded FPU instruction.
APUFCMDECLDSTXFERSIZE[0:2]	Output	This bus indicates the APU controller decoded load/store transfer size. • `100`: Byte • `010`: Halfword • `001`: Word • `011`: Doubleword • `111`: Quadword
APUFCMDECLOAD	Output	A High on this output indicates an APU controller decoded load instruction.
APUFCMDECNONAUTON	Output	This signal is asserted to indicate the presence of an APU controller decoded instruction that is a non-autonomous instruction (this includes Store instructions because stores return data to the CPU).
APUFCMDECSTORE	Output	A High on this output indicates an APU controller decoded store instruction.
APUFCMDECUDI[0:3]	Output	This bus specifies the decoded UDI register.
APUFCMDECUDIVALID	Output	This signal is asserted to indicate APUFCMDECUDI[0:3] is valid.
APUFCMENDIAN	Output	This signal indicates the setting of the load/store endian attribute. • `0`: big endian • `1`: little endian
APUFCMFLUSH	Output	This signal is asserted to flush the FCM instruction.
APUFCMINSTRUCTION[0:31]	Output	This bus contains the instruction presented to the FCM.
APUFCMINSTRVALID	Output	This signal is asserted to indicate the instruction on APUFCMINSTRUCTION[0:31] is valid.
APUFCMLOADBYTEADDR[0:3]	Output	This bus specifies at which of the 16 bytes the data begins for the load transfer.
APUFCMLOADDATA[0:127]	Output	This 128-bit bus contains load data.
APUFCMLOADDVALID	Output	This signal is asserted to indicate APUFCMLOADDATA[0:127] is valid.
APUFCMMSRFE0	Output	This signal indicates the value of MSR[FE0]. It is used for FPU instructions only.
APUFCMMSRFE1	Output	This signal indicates the value of MSR[FE1]. It is used for FPU instructions only.
APUFCMNEXTINSTRREADY	Output	This signal is asserted to indicate the APU controller will send the next autonomous instruction along with all data on the clock cycle after the current FCM instruction is finished (when FCMAPUDONE is asserted).
APUFCMOPERANDVALID	Output	This signal is asserted to indicate the instruction operands are valid.
APUFCMRADATA[0:31]	Output	This bus contains the instruction operand from GPR(Ra).

Table 12-1: **APU to FCM Signal Descriptions *(Cont'd)***

Interface Signal	Direction	Function
APUFCMRBDATA[0:31]	Output	This bus contains the instruction operand from GPR(Rb).
APUFCMWRITEBACKOK	Output	This signal is asserted to indicate it is safe for the FCM to commit internal state changes.
FCMAPUCONFIRMINSTR	Input	This signal is asserted to indicate the FCM does not cause an exception for this instruction. This signal is used for non-autonomous operations with late confirmation.
FCMAPUCR[0:3]	Input	This bus contains the condition record bits for the CR field, specified by the instruction.
FCMAPUDONE	Input	This signal is asserted to indicate completion of the FCM instruction in the APU controller.
FCMAPUEXCEPTION	Input	This signal is asserted to indicate an FCM generated program exception. The exception must be enabled by the processor to trap.
FCMAPUFPSCRFEX	Input	This signal is asserted to indicate an FPU instruction generated an exception. The level on this signal should reflect the value of the FPSCR[FEX] bit in the FPU.
FCMAPURESULT[0:31]	Input	This bus contains the FCM execution result, which is passed to the CPU through the APU controller.
FCMAPURESULTVALID	Input	When this signal is asserted, values on FCMAPURESULT[0:31], FCMAPUCR[0:3], or FCMAPUSTOREDATA[0:127] are valid.
FCMAPUSLEEPNOTREADY	Input	This signal is asserted to indicate the FCM is still executing an instruction. This signal determines when the CPU is allowed to enter sleep mode. The APU Controller prevents the CPU from requesting sleep mode while an instruction is inside the APU controller. The FCM can use this signal to extend this time after an instruction has completed in the APU, but this signal must not be tied High. If not used, this signal must be tied Low.
FCMAPUSTOREDATA[0:127]	Input	This 128-bit bus contains separate store data.

Instruction Decoding

The processor presents up to two instructions to the APU controller during the Pre-Decode stage. The APU controller can decode up to two FCM instructions in the same cycle. It can also decode all FPU instructions (except the "extended" load/store instructions) and up to 16 UDIs, FCM loads and stores, and VMX loads and stores. The APU controller decodes the instructions to notify the processor what resources the instruction requires (for example, if the instruction is a load, its transfer size, and any source data needed). The APU controller also generates certain decode signals for the FCM to ease the FCM decode logic.

With the exception of some FPU instructions, FCM instructions conform to the general format shown in Figure 12-3.

0 — 5	6 — 10	11 — 15	16 — 20	21 — 31
Primary Opcode	RT	RA	RB	Extended Opcode

Figure 12-3: **FCM Instruction Format**

The processor uses both primary and extended opcodes to identify potential FCM instructions. The opcodes are decoded by the APU controller to identify uniquely the specific FCM instruction resource needs. Generally, the RA and RB fields specify operand registers, and the RT field specifies the target register. UDIs can be configured to interpret these bit fields differently. For example, the five-bit fields can be used for immediate values. UDIs can also be configured to allow for five "wildcard" bits in the extended opcode. When in wildcard mode, bits [21:25] of the instruction can be used in any way. For example, the user can use these five bits as another FCM register (but not as a CPU register value), a 5-bit immediate value, or to configure a group of instructions using only one UDI register.

FPU Instructions

The APU controller can be enabled to decode for all the floating-point instructions (except the "extended" load/store instructions) when an FPU is attached on the FCM interface. Refer to the *Book E: Enhanced PowerPC Architecture Specification* [Ref 1] for detailed information about the floating-point instructions. The APU controller can selectively disable the following six groups of floating-point instructions:

- complex arithmetic
- conversion
- estimate/select
- FPSCR
- single-precision only
- double-precision only

Table 12-2 lists the instructions in these groups.

Table 12-2: **Floating-Point Instructions by Group**

Complex Arithmetic Group					
fdiv	fdiv.	fdivs	fdivs.	fsqrt	fsqrt.
fsqrts	fsqrts.				
Conversion Group					
fcfid	fctid	fctidz	fctiw	fctiw.	fctiwz
fctiwz.	frsp	frsp.			
Estimate/Select Group					
fres	fres.	frsqrte	frsqrte.	fsel	fsel.
FPSCR Group					
mcrfs	mffs	mffs.	mtfsb0	mtfsb0.	mtfsb1
mtfsb1.	mtfsf	mtfsf.	mtfsfi	mtfsfi.	
Single-Precision Only Group					
lfs	lfsu	lfsx	lfsux	stfs	stfsu
stfsx	stfsux	fadds	fadds.	fsubs	fsubs.
fdivs	fdivs.	fmuls	fmuls.	fsqrts	fsqrts.
fmadds	fmadds.	fnmadds	fnmadds.	fmsubs	fmsubs.

Table 12-2: **Floating-Point Instructions by Group *(Cont'd)***

fnmsubs	fnmsubs.				
Double-Precision Only Group					
lfd	lfdu	lfdx	lfdux	stfd	stfdu
stfdx	stfdux	stfiwx	fadd	fadd.	fsub
fsub.	fdiv	fdiv.	fmul	fmul.	fsqrt
fsqrt.	fmadd	fmadd.	fnmadd	fnmadd.	fmsub
fmsub.	fnmsub	fnmsub.			

The APU controller also provides decode signals to the FCM/FPU. These signals include the following information:

- Whether the instruction is an FPU instruction
- Whether the instruction is a Load or Store instruction
- The size of the transfer (if the instruction is a Load or a Store)
- Whether the instruction is a non-autonomous instruction (includes store instructions)

FCM User-Defined Instructions

The user can configure up to 16 UDI registers to be decoded by the APU controller. The UDIs conform to the standard FCM instruction format. The interpretation of the RA, RB, and RT fields are up to the FCM. In other words, the FCM can use the separate five-bit fields as the registers in the processor's GPR, as immediate values, as internal FCM registers, or for some other purpose. The specific primary and extended opcodes that UDIs can use are shown in Table 12-3.

Table 12-3: **Primary and Extended Opcodes**

Primary Opcode [0:5]	Extended Opcode [21:31]	Description
0 (= `0b000000`)	`0b00000000000`	Illegal
	All except above	Available for UDIs that do not set the CR bits
4 (= `0b000100`)	`0b------1--0-`	MAcc and Xilinx reserved
	`0b1----000110`	Available for UDIs that do need to set the CR bits
	All except above	Available for UDIs that do not set the CR bits

The user also can "wildcard" some of the extended opcode. When a wildcard is set for a particular UDI, the extended opcode bits [21:25] can be used as the user wishes. Thus the user can use these five bits as immediate values, an internal FCM register, or to define a group of instructions that have the same extended opcode bits [26:31]. When using "wildcard" mode for instructions, the user must follow these restrictions:

1. No other UDI can be configured with the same primary opcode and extended opcode bits [26:31].

2. All instructions in the same UDI group must use the same options. In other words, the group must all be autonomous, must all use Ra source operand, must all be non-autonomous with early confirm, and so on.

UDIs are configured using UDI Configuration registers, which can be accessed through the DCR interface.

Any processor resources needed for the UDI are defined in the APU controller UDI registers as well as in the APU Controller Configuration register. These two types of registers are explained in detail in "APU Configuration," page 205.

When a UDI is decoded by the APU controller, the FCM receives decoded information along with the 32-bit instruction. The decode signals include the following information:

- Bit encoded UDI register number (`4'h0` = UDI0, `4'h1` = UDI1, and so on)
- Valid bit for the UDI register number
- Whether the instruction is a non-autonomous instruction

FCM Load/Store Instructions

The APU has the ability to decode and issue FCM load and store instructions, which allow the transfer of data between the processor's memory system and the Fabric Coprocessor Module (FCM). The processor handles the address calculation and also passes data to/from the memory. An FCM load transfers data from a memory location to a destination register in the FCM and vice-versa for an FCM store. An FCM load/store can be of size byte, halfword, word, doubleword, or quadword. The FCM load/store can also be of type Update or not. Update capable instructions update the base address register RA with the calculated effective address. Figure 12-4 shows the format of the FCM load/store instructions.

Primary Opcode [0:5]	Extended Opcode [21:31]										
0 1 1 1 1 1	U	W_0	L/S	W_1	W_2	0	0	1	1	1	0

Figure 12-4: **FCM Load/Store Instruction Format**

Table 12-4: **FCM Load/Store Instruction Encoding**

Bit	Description
U	Update Capability. `0`: Not update capable `1`: Update capable
$W_{[0:2]}$	Size. `000`: Byte `001`: Halfword `010`: Word `x11`: Quadword `100` Doubleword `101`: Invalid `110`: Invalid
L/S	Load/Store. `0`: Load `1`: Store

The APU controller also provides decode signals to the FCM for load/store instructions. These signals contain the following information:

- Load instruction
- Store instruction
- Size of transfer
  - `100` = byte
  - `010` = halfword
  - `001` = word
  - `011` = doubleword
  - `111` = quadword
- Non-autonomous instruction (in the case of a store)

Instruction Execution

There are two major classes of FCM instructions: storage (loads and stores) and non-storage. The storage instructions are more rigidly defined and are tightly coupled with the processor pipeline. The non-storage instructions have more flexibility as to their opcodes and their function.

Storage Instructions (FCM Loads and Stores)

The FCM can execute loads and stores in bytes, halfwords, words, doublewords, and quadwords. The processor executes the address calculation for any load or store and also passes the data to/from memory. The processor also replaces the base address with the effective address when executing any load or store with an update. Load instructions are considered to be autonomous (they do not stall the processor pipeline until finished and do not return data to the processor), and store instructions are considered to be non-autonomous (they stall the processor pipeline until the store data is returned).

All load and store data must be contained within a quadword boundary. For example, a quadword load must have an address aligned on the quadword boundary (byte 0 of the 128 bits), but a doubleword load can have a starting address at byte 0, 1, 2, 3, 4, 5, 6, 7, or 8. For load data, the APU Controller sends the entire 128-bit bus along with the starting byte address to the FCM. The FCM must look at the starting byte address to determine where in the 128 bits the valid data resides. For stores, the data should be returned in the most significant bits. For example, a store with a byte length should be on bits [0:7], a word on bits [0:31], and so on. The APU Controller steers the data correctly to the processor.

The APU controller supports a 128-bit load bus and a 128-bit store bus, allowing for the transfer of a full quadword in one FCM clock cycle. In Virtex-4 FX devices, this transfer took four clock cycles over a 32-bit bus (see [Ref 8] for more information). The APU controller accepts up to three loads or stores from the processor at a time, which allows the APU controller, for example, to buffer a second and third quadword while it is sending the first quadword to the FCM. The second quadword is then ready to send to the FCM immediately following the first transfer and overlaps any overhead needed in receiving the data of the second transfer from the processor. Figure 12-5 is a simple block diagram of the load data flow in the APU controller.

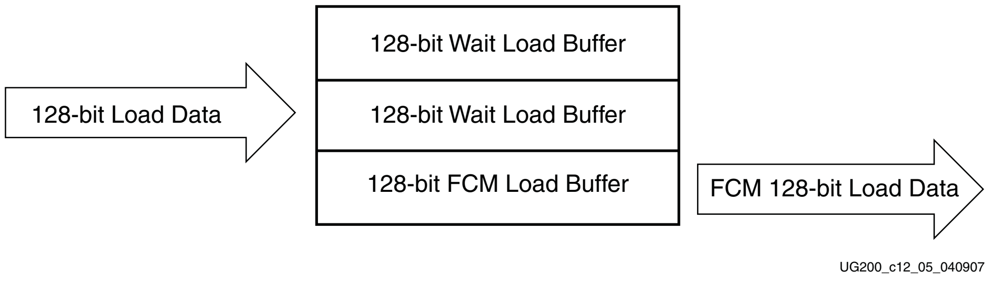


Figure 12-5: **APU Controller Load Data Flow**

If the FCM load/store instruction is flushed from the processor pipeline, the APU controller notifies the FCM by sending a Flush signal. The load/store instruction can be flushed from the pipeline because of an address alignment exception, a TLB miss, an access with an endian attribute not supported by the hardware, or if a previous instruction causes the pipeline to flush. Similar to the Flush signal, the APU controller provides the APUFCMWRITEBACKOK signal, which indicates when an FCM load instruction can no longer be flushed and can safely update its internal registers. This signal is optional for store instructions (this signal can cause a performance hit for stores).

The APUFCMNEXTINSTRREADY signal can be used to improve instruction throughput for load instructions. This signal is asserted High when the next load instruction and all of its data are ready to be sent to the FCM. A High on this signal means that as soon as the APU controller receives an asserted FCMAPUDONE signal for the current instruction, the next load instruction with all data is sent on the next FCM cycle. Because the FCM knows when the next instruction will arrive, it can assert FCMAPUDONE High during that same cycle. Thus the load instruction can be sent and completed in one FCM clock cycle.

Load Execution Details

Load instructions follow a specific sequence of signals on the FCM interface. This description applies for both APU and FPU loads. The FCM receives the following signals when the instruction is sent:

- APUFCMINSTRVALID = 1
- APUFCMDECFPUOP = 1 if FPU or 0 if APU
- APUFCMDECLOAD = 1
- APUFCMDECLDSTXFERSIZE[0:2] = `100` (byte), 010 (halfword), `001` (word), `011` (doubleword), or `111` (quadword)
- APUFCMINSTRUCTION[0:31] = the 32-bit instruction

During the same cycle, the instruction signals initially go High or at a later cycle the FCM receives the following signals:

- APUFCMLOADDVALID = 1
- APUFCMLOADDATA[0:127] = load data bus
- APUFCMLOADBYTEADDR[0:3] = starting byte address of the data within the 16 bytes
- APUFCMENDIAN = 1 if little endian format or 0 if big endian format

All of the above signals remain valid until the transaction is complete. The transaction is complete when one of the following occurs:

- The FCM received a pulse of APUFCMWRITEBACKOK = 1 and then sends back FCMAPUDONE = 1 during the same cycle or sometime after receiving either APUFCMWRITEBACKOK or APUFCMLOADDVALID (whichever is later)

  or

- If the instruction is flushed from the processor pipeline, it receives APUFCMFLUSH = 1.

If the latter occurs the FCM should not return FCMAPUDONE = 1 for this instruction.

Store Execution Details

There are two main types of store instructions: stores using *writebackok* and stores that do not need or want *writebackok*. Stores without *writebackok* often have better performance. Both types can be used with either APU or FPU stores. To set the store to use *writebackok*, bit 16 of the APU Control Register bit must be set to 1. Both types of stores begin the same way:

- APUFCMINSTRVALID = 1
- APUFCMDECFPUOP = 1 if FPU or 0 if APU
- APUFCMDECSTORE = 1
- APUFCMDECLDSTXFERSIZE[0:2] = `100` (byte), `010` (halfword), `001` (word), `011` (doubleword), or `111` (quadword)
- APUFCMINSTRUCTION[0:31] = the 32-bit instruction
- APUFCMDECNONAUTON = 1

For stores that do not need to wait for APUFCMWRITEBACKOK to be asserted (if no resources in the FCM are updated based on the store completing):

- FCMAPURESULTVALID = 1, set High when the store data bus is valid
- FCMAPUSTOREDATA[0:127] = store data bus, data should be in the most-significant bits of the bus (for example, a byte transfer at FCMAPUSTOREDATA[0:7] or a word transfer at FCMAPUSTOREDATA[0:31])
- FCMAPUDONE = 1, set High in the same cycle or after FCMAPURESULTVALID

Note: If the store instruction is flushed before FCMAPUDONE is asserted, the APU Controller can assert APUFCMFLUSH for a store that does not use *writebackok*. In this case, the FCM should not assert FCMAPUDONE.

For stores that wait for APUFCMWRITEBACKOK (either resources in the FCM are updated when the store completes, or the FCM could have a different endianess than the memory and needs to wait for APUFCMENDIAN), the FCM must wait until the following signals are received:

- APUFCMWRITEBACKOK = 1 (pulsed), if the store will complete
- APUFCMENDIAN = 1 (little endian) or 0 (big endian), and valid the same cycle as APUFCMWRITEBACKOK
- APUFCMFLUSH = 1 if the store was flushed from the processor pipeline (the FCM does not receive APUFCMWRITEBACKOK)

When the FCM receives an asserted APUFCMWRITEBACKOK, it can send the following signals in the same or any later cycle:

- FCMAPURESULTVALID = 1, set High when the store data bus is valid
- FCMAPUSTOREDATA[0:127] = store data bus, data should be in the most-significant bits of the bus (for example, a byte transfer at FCMAPUSTOREDATA[0:7] or a word transfer at FCMAPUSTOREDATA[0:31])
- FCMAPUDONE = 1, set High the same cycle or after FCMAPURESULTVALID is asserted

For more details on the signal timing, refer to "Timing Diagrams for the APU Controller," page 214.

Non-Storage Instructions

The APU controller supports three execution modes: autonomous, non-autonomous with early confirmation, and non-autonomous with late confirmation.

- Autonomous Instructions

  Instructions in the autonomous class do not stall the processor pipeline. They are typically fire-and-forget type instructions that do not return any result data or condition record bits to the processor. The FCM cannot generate an exception for this class of instruction. The APU controller automatically confirms to the processor that this instruction will not cause an exception, which allows the FCM to receive an asserted APUFCMWRITEBACKOK signal (when the instruction can no longer be flushed from the processor pipeline) as soon as possible. An example of an autonomous instruction is an instruction that reads the contents of two general-purpose registers (GPRs) without returning any data to the processor.

- Non-autonomous with Early Confirmation Instructions

  Instructions in the non-autonomous class stall normal execution in the processor pipeline until the FCM instruction is completed. These instructions can return result data and/or status (condition record bits) to the processor. For Non-autonomous with Early Confirmation instructions, the FCM cannot generate an exception. The APU controller automatically confirms to the processor that this instruction will not cause an exception, which allows the FCM to receive the APUFCMWRITEBACKOK signal (when the instruction can no longer be flushed from the processor pipeline) as soon as possible.

- Non-autonomous with Late Confirmation Instructions

  Instructions in the non-autonomous class stall normal execution in the processor pipeline until the FCM instruction is completed. These instructions can return result data and/or status (condition record bits) to the processor. For Non-autonomous instructions with Late Confirmation, the APU controller waits for the FCM to confirm that this instruction does not cause an exception (FCMAPUCONFIRMINSTR), which allows the FCM to throw a precise exception for the instruction, if necessary. It also causes the APUFCMWRITEBACKOK signal to arrive somewhat later than in the case of Early Confirmation. This instruction type has not been optimized for performance; however, this instruction type does allow the FCM to generate its own precise instruction exception.

  There are no other instruction types that support FCM generated precise exceptions. FPU instructions in the FPSCR group are always executed as Non-autonomous with Late Confirmation instructions. Also, when MSR[FE0] or MSR[FE1] is set to 1, all non-storage FPU instructions execute as Non-autonomous with Late Confirmation instructions.

Non-Storage Instruction Execution

Non-storage instructions can use source data from the GPR in the processor, send result data to the GPR in the processor, and update Condition Record (CR) bits in the processor. The APU controller does not allow the FCM to return Carry or Overflow data to the processor.

For non-storage instructions that return data back to the processor (result data or CR bits), FCMAPURESULTVALID must assert High at least one clock cycle before FCMAPUDONE. However, for FPU/APU store instructions that also return the data back to the processor, FCMAPURESULTVALID and FCMAPUDONE can assert High in the same clock cycle.

To improve performance, the APU controller accepts up to three FCM instructions at a time from the processor, allowing the APU controller, for example, to finish the first FCM instruction at the same time it receives the source data for the second and third FCM instructions. The second instruction then has all of its source data ready to send once the first FCM instruction has finished and overlaps the overhead of starting the second and third instructions in the processor pipeline.

If the FCM non-storage instruction is flushed from the processor pipeline, the APU controller notifies the FCM by sending a Flush signal. The instruction can be flushed from the pipeline because of an FCM generated exception or if a previous instruction causes the pipeline to flush. Similar to the Flush signal, the APU controller provides a signal indicating when an FCM non-storage instruction can no longer be flushed and can safely update its internal registers (APUFCMWRITEBACKOK).

The APUFCMNEXTINSTRREADY signal can be used to improve instruction throughput for autonomous instructions. This signal is driven High when the next autonomous instruction and all of its data (if any) are ready to be sent to the FCM. A High on this signal means that as soon as the APU controller receives an asserted FCMAPUDONE signal for the current instruction, the next autonomous instruction with all data is sent on the very next FCM cycle. Because the FCM knows when the next instruction will arrive, it can drive FCMAPUDONE High during that same cycle, allowing the autonomous instruction to be sent and completed in one FCM clock cycle.

Exceptions

There are three main scenarios when exceptions occur due to UDI or FPU instructions:

- Storage exceptions (in the case of UDI or FPU load/store instructions)
- Exceptions generated by the APU controller decoder (for example, when decoding is disabled)
- FCM generated exceptions

Storage Exceptions

The processor generates storage exceptions for UDI and FPU load/store instructions under certain circumstances because the processor handles the address calculation, TLB access, and cache and/or memory access. The following exceptions might occur when executing a UDI or FPU load/store instruction:

- Read Access Control Exception

While in user mode (MSR[PR] = 1), a load instruction attempts to access a location in memory that is not enabled for read access in user mode. While in supervisor mode (MSR[PR] = 0), a load instruction attempts to access a location in memory that is not enabled for read access in supervisor mode.

- Write Access Control Exception

  While in user mode (MSR[PR] = 1), a store instruction attempts to access a location in memory that is not enabled for write access in user mode. While in supervisor mode (MSR[PR] = 0), a store instruction attempts to access a location in memory that is not enabled for write access in supervisor mode.

- Byte Ordering Exception

  This exception, which is indicated by the Endian attribute bit, occurs when the attached FCM does not support the current byte ordering of the memory. When a load/store instruction is executed with TrapBE (APU Control register bit [21]) or TrapLE (APU Control register bit [22]), this exception might occur.

- Data TLB Error Interrupt

  This exception occurs when a load/store instruction attempts to access a virtual address for which a valid TLB entry does not exist.

- Alignment Interrupt

  This exception occurs when a load/store instruction references a data storage operand that crosses a quadword boundary.

APU Controller Decode Exceptions

The APU controller can cause exceptions when the APU Control register has been configured to disable certain instruction decoding. The following exceptions might occur when executing a UDI or FPU instruction:

- Floating-Point Unavailable Interrupt

  This exception occurs when an attempt is made to execute a floating-point instruction that is recognized by the APU controller (FCM Enable = 1 and FPU Decode Disable = 0) and MSR[FP] = 0.

- Illegal Instruction Exception

  This exception occurs when there is an attempt to execute the following:

  - a UDI or FPU instruction and FCM Enable = 0
  - an FPU instruction and FPU Decode Disable = 1
  - a UDI instruction and UDI Decode Disable = 1
  - an FCM Load/Store instruction and Load/Store Decode Disable = 1

- Unimplemented Operation Exception

  This exception occurs when an attempt is made to execute the following:

  - an instruction in the FPU Complex Arithmetic group and MSR[FP] = 1, FPU Decode Disable = 0, and FPU complex arithmetic Disable = 1.
  - an instruction in the FPU Convert group and MSR[FP] = 1, FPU Decode Disable = 0, and FPU convert. Disable = 1.
  - an instruction in the FPU Estimate/Select group and MSR[FP] = 1, FPU Decode Disable = 0, and FPU estimate/select Disable = 1.

- an instruction in the FPU FPSCR group and MSR[FP] = 1, FPU Decode Disable = 0, and FPU FPSCR Disable = 1.
- an instruction in the FPU single-precision only group and MSR[FP] = 1, FPU Decode Disable = 0, and FPU single-precision Disable = 1.
- an instruction in the FPU Double-precision only group and MSR[FP] = 1, FPU Decode Disable = 0, and FPU double-precision Disable = 1.

- Privileged Instruction Exception

  This exception occurs when MSR[PR] = 1 (user mode) and an attempt is made to execute a UDI instruction that is privileged.

FCM Generated Exceptions

The FCM can also generate precise exceptions. To generate precise exceptions, the instruction must be non-autonomous with late confirm. Because of this restriction, no load/store instructions can cause a precise exception generated by the FCM. If the FCM wishes to generate an exception for a different type of instruction, the FCM should generate an external interrupt to the processor. The following exceptions can be generated by the FCM:

- Floating-Point Enabled Exception

  This exception occurs when the execution or attempted execution of a recognized floating-point instruction causes FPSCR[FEX] to be set to 1. The floating-point instruction must be executed as non-autonomous with late confirm. If MSR[FE0, FE1] are non-zero, a precise Program Interrupt occurs. If MSR[FE0, FE1] are zeros, the instruction completes normally. When MSR[FE0, FE1] become non-zero and FPSCR[FEX] is still set to 1, a “delayed” or imprecise Program Interrupt occurs. When MSR[FE0, FE1] is non-zero, all FPU instructions are forced to be of type non-autonomous with late confirm to keep the FPU generated exceptions precise. When MSR[FE0, FE1] is zero, only the FPSCR instructions are implemented as non-autonomous with late confirmation to allow for imprecise interrupts.

- Auxiliary Processor Enabled Exception

  This exception occurs during the attempted execution of a UDI of type non-autonomous with late confirm. If the FCM drives FCMAPUEXCEPTION High instead of asserting FCMAPUCONFIRMINSTR, the precise exception occurs.

FPU Generated Exception Execution Details

FPU generated exceptions must follow a strict sequence. As stated before, the instruction must be a non-autonomous instruction with late confirm. FPU instructions can only generate an exception when FPSCR[FEX] is set to 1. After the instruction has been sent to the FPU the following sequence should occur:

- FCMAPUEXCEPTION = 1 (held, not pulsed)
- FCMAPUFPSCRFEX = 1 (held, not pulsed)

When the APU controller sees these signals, it responds with:

- APUFCMWRITEBACKOK =1 if the FPU exception was processed/accepted, 0 if not
- APUFCMFLUSH = 1 if a previous exception was received, 0 if the FPU exception is accepted

The transaction is complete when either the FPU receives an asserted APUFCMFLUSH or when the FPU returns FCMAPUDONE = 1 after receiving APUFCMWRITEBACKOK = 1.

If APUFCMFLUSH is asserted, the FPU must deassert FCMAPUEXCEPTION and FCMAPUFPSCRFEX (FPSCR[FEX] must not be updated in the FPU because the instruction was flushed). If APUFCMWRITEBACKOK is asserted, the FPU can deassert FCMAPUEXCEPTION when sending FCMAPUDONE or leave it High to be cleared later by software. If FCMAPUEXCEPTION is left High, any later non-autonomous instruction with late confirm is seen as causing an exception. FCMAPUFPSCRFEX should remain High until FPSCR[FEX] has been cleared by software.

APU Generated Exception Execution Details

APU generated exceptions follow a similar sequence to FPU exceptions but they are simpler. Again, only non-autonomous instructions with late confirm can generate an exception. After the FCM has received the instruction the following sequence occurs:

- FCMAPUEXCEPTION = 1 (held, not pulsed)

When the APU controller sees these signals, it responds with:

- APUFCMWRITEBACKOK = 1 if the APU exception was processed/accepted, 0 if not
- APUFCMFLUSH = 1 if a previous exception was received and 0 if the APU exception is accepted

The transaction is complete when either the FCM receives an asserted APUFCMFLUSH or when the FCM returns FCMAPUDONE = 1 after receiving APUFCMWRITEBACKOK = 1. If an asserted APUFCMFLUSH is received, the FCM must deassert FCMAPUEXCEPTION. If an asserted APUFCMWRITEBACKOK is received, the FCM can deassert FCMAPUEXCEPTION when sending FCMAPUDONE or leave it High to be cleared later by software. If FCMAPUEXCEPTION is left High, any later non-autonomous instruction with late confirm is seen as causing an exception.

For more details on the signal timing, refer to "Timing Diagrams for the APU Controller," page 214.

APU Configuration

Enabling the APU Controller

The MSR register must be configured before the processor can use the APU controller. Table 12-5 describes the APU controller-related bits in the MSR.

Table 12-5: **APU Controller-Related MSR Bits**

Bit(s) in MSR	Description
18	FCM floating-point unit present • 1: true • 0: false
(20,23)	Floating-point exception mode (FE0,FE1): • (0,0): Ignore floating-point exceptions • (1,0): Imprecise recoverable mode • (0,1): Imprecise non-recoverable mode • (1,1): Precise mode

Configuration Registers

The APU controller is configured through a single 32-bit APU Configuration register and 16 32-bit UDI registers.

DCR Access to Configuration Registers

The APU and UDI configuration registers are accessed through the DCR interface. The 16 UDI registers share the same DCR address. Figure 12-6 shows the DCR access of UDI registers.

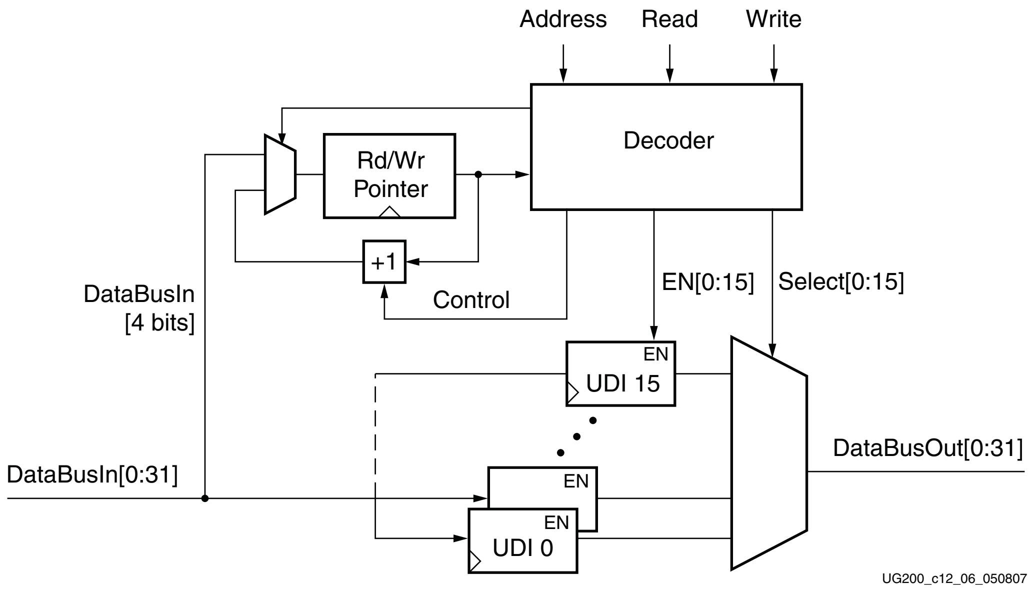


Figure 12-6: **DCR Access of UDI Registers**

A DCR read from or write to the UDI configuration register address uses a 4-bit read/write pointer register in the APU controller to select which specific UDI configuration to read or write. This pointer auto-increments after each DCR read or write operation. To load the read/write pointer with a specific value, the user must perform a *ghost* write to the UDI configuration DCR address. This write does not affect the contents of any UDI configuration registers, only the read/write pointer.

A DCR read performed to the UDI configuration address after a ghost write returns the contents of the desired UDI configuration register, and a DCR write writes to the desired UDI configuration register.

Refer to Chapter 7, "Device Control Register Bus," for more information on programmatically accessing these configuration registers.

APU Control Register

The APU Control register turns on or off various features in the APU controller. Figure 12-7 shows the bits in the APU Control register. Table 12-6 defines the bits within the register.

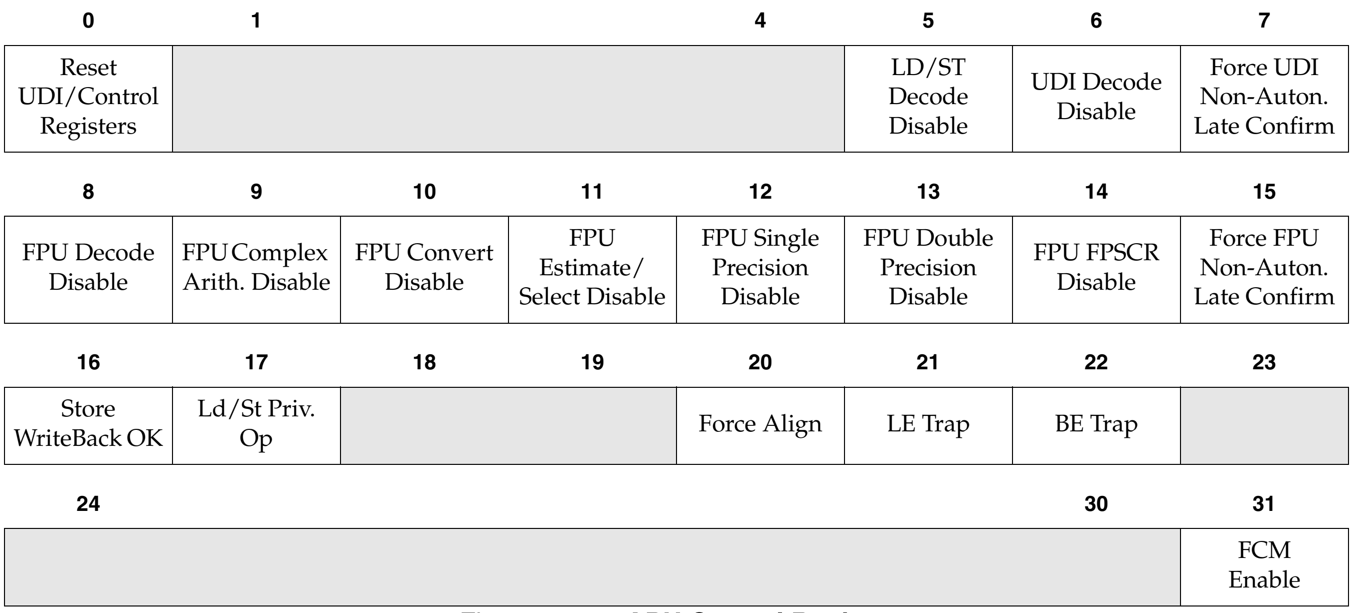

0
1
4
5
6
7
Reset UDI/Control Registers
LD/ST Decode Disable
UDI Decode Disable
Force UDI Non-Auton. Late Confirm
8
9
10
11
12
13
14
15
FPU Decode Disable
FPU Complex Arith. Disable
FPU Convert Disable
FPU Estimate/ Select Disable
FPU Single Precision Disable
FPU Double Precision Disable
FPU FPSCR Disable
Force FPU Non-Auton. Late Confirm
16
17
18
19
20
21
22
23
Store WriteBack OK
Ld/St Priv. Op
Force Align
LE Trap
BE Trap
24
30
31
FCM Enable

Figure 12-7: **APU Control Register**

Table 12-6: **Bit Descriptions for the APU Control Register**

Bit	Name	Default Value	Description
0	Reset UDI/Control Registers	-	When a 1 is written to this bit, all the UDI registers are reset to their default values. The rest of the bits in the control register are also reset to their default values. When read, this bit always returns a 0.
1:4	Reserved	-	Reserved
5	LD/ST Decode Disable	0	When set, this bit disables all FCM Load/Store decoding in the APU controller. This does not affect FPU Load/Store instructions. An FCM Load/Store in the program causes an illegal instruction exception.
6	UDI Decode Disable	0	When set, this bit disables all UDI decoding in the APU controller. This does not affect FCM Load/Store or FPU instructions. A UDI instruction in the program causes an illegal instruction exception.
7	Force UDI Non-Autonomous, Late Confirm	0	When set, this bit forces any non-storage UDI instruction to be executed as a non-autonomous instruction with late confirm regardless of the type indicated in the UDI register.
8	FPU Decode Disable	1	When set, this bit disables all FPU decoding in the APU controller. An FPU instruction in the program causes an illegal instruction exception.
9	FPU Complex Arithmetic Disable	0	When set, this bit disables decoding for all FPU divide and square root instructions (fdiv, fdiv., fdivs, fdivs., fsqrt, fsqrt., fsqrts, fsqrts.). An FPU complex arithmetic instruction in the program when FPU Decode is not disabled causes an unimplemented instruction exception.

Table 12-6: **Bit Descriptions for the APU Control Register *(Cont'd)***

Bit	Name	Default Value	Description
10	FPU Convert Disable	0	When set, this bit disables decoding for all FPU convert instructions (fcfid, fctid, fctidz, fctiw, fctiw., fctiwz, fctiwz., frsp, frsp.). An FPU convert instruction in the program when FPU Decode is not disabled causes an unimplemented instruction exception.
11	FPU Estimate/select Disable	0	When set, this bit disables decoding for all FPU estimate instructions (fres, fres., frsqrte, frsqrte., fsel, fsel.). An FPU estimate instruction in the program when FPU Decode is not disabled causes an unimplemented instruction exception.
12	FPU Single Precision Disable	0	When set, this bit disables decoding for all FPU single-precision only instructions (lfs, lfsu, lfsx, lfsux, stfs, stfsu, stfsx, stfsux, fadds, fadds., fsubs, fsubs., fdivs, fdivs., fmuls, fmuls., fsqrts, fsqrts., fmadds, fmadds., fnmadds, fnmadds., fmsubs, fmsubs., fnmsubs, fnmsubs.). A single-precision FPU instruction in the program when FPU Decode is not disabled causes an unimplemented instruction exception.
13	FPU Double Precision Disable	0	When set, this bit disables decoding for all FPU double-precision only instructions (lfd, lfdu, lfdx, lfdux, stfd, stfdu, stfdx, stfdux, stfiwx, fadd, fadd., fsub, fsub., fdiv, fdiv., fmul, fmul., fsqrt, fsqrt., fmadd, fmadd., fnmadd, fnmadd., fmsub, fmsub., fnmsub, fnmsub.). If a double-precision FPU instruction is encountered in the program and FPU Decode is not disabled, an unimplemented instruction exception occurs.
14	FPU FPSCR Disable	0	When set, this bit disables decoding for all FPSCR FPU instructions (mcrfs, mffs, mffs., mtfsb0, mtfsb0., mtfsb1, mtfsb1., mtfsf, mtfsf., mtfsfi, mtfsfi.). An FPSCR instruction in the program when FPU Decode is not disabled causes an unimplemented instruction exception.
15	Force FPU Non-Autonomous, Late Confirm	0	When set, this bit forces all non-storage FPU instructions to be executed as non-autonomous instructions with late confirm.
16	Store WritebackOK	0	When this bit is set, the APU controller waits to send a WritebackOK signal to the FCM for all store instructions (both APU and FPU stores). The WritebackOK signal is sent after the store instruction passes the LWB stage in the CPU pipe, which can cause a slight performance hit when executing store instructions.
17	LD/ST Privilege	0	When this bit is set, any load or store UDI executes in privileged mode (this does not affect FPU load/store instructions).
18:19	Reserved	-	Reserved
20	Force Align	0	When this bit is set, any load or store (both APU and FPU) forces alignment. The address is forced to align on the natural boundary of the transfer (word boundary for a word transfer, doubleword boundary for a doubleword transfer, and so forth). This also prevents an alignment exception.
21	LE Trap	0	When this bit is set, any load or store (both APU and FPU) traps when the Endian storage attribute is `1'b1` (little Endian).
22	BE Trap	0	When this bit is set, any load or store (both APU and FPU) traps when the Endian storage attribute is `1'b0` (big Endian).

Table 12-6: **Bit Descriptions for the APU Control Register *(Cont'd)***

Bit	Name	Default Value	Description
23:30	Reserved	-	Reserved
31	FCM Enable	0	When this bit is set, the FCM interface is enabled and the APU controller decodes instructions. When this bit is cleared, bits 5, 6, and 8 are overridden. The APU controller does not decode any instructions.

User-Defined Instruction (UDI) Configuration Registers

For all UDIs, the user needs to configure the primary and extended opcodes along with any necessary execution options. Figure 12-8 shows the UDI Configuration register bits. Table 12-7 defines the bits in the UDI Configuration register.

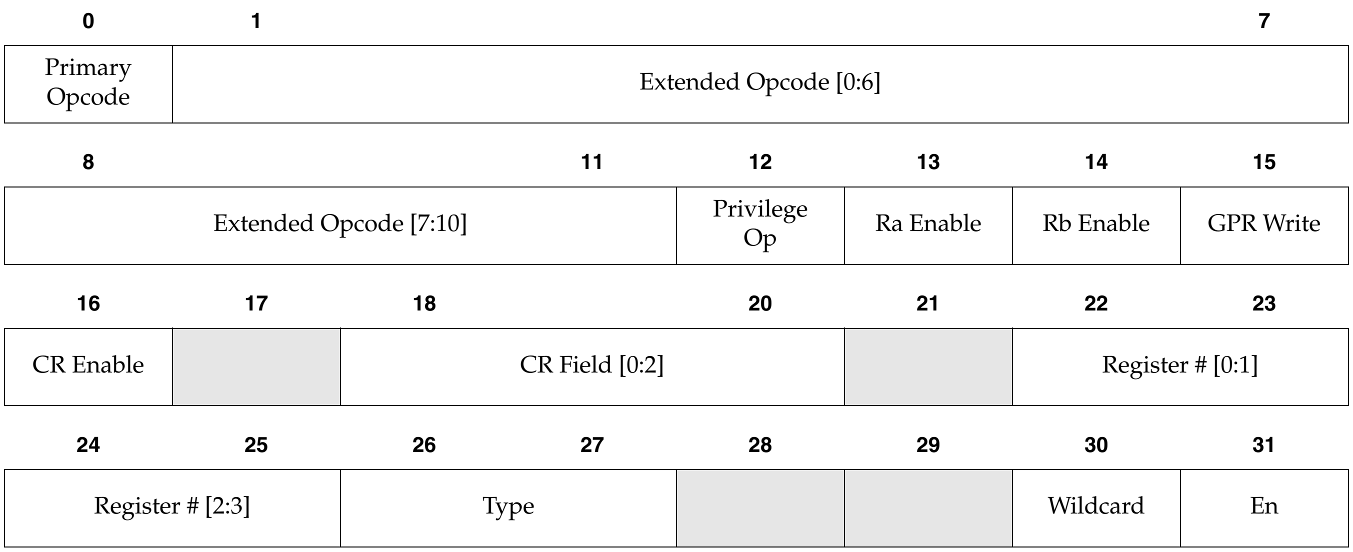


Figure 12-8: **UDI Configuration Register**

Table 12-7: **Bit Descriptions for the UDI Configuration Register**

Bit	Name	Description
0	Primary Opcode	• 0: `6'b000000` (opcode 0) • 1: `6'b000100` (opcode 4)
1:11	Extended Opcode	11 bits of the full extended opcode
12	Privilege Op	When this bit is set, this instruction must execute in privilege mode.
13	Ra Enable	When this bit is set, this instruction needs to read the Ra source operand from the GPR.
14	Rb Enable	When this bit is set, this instruction needs to read the Rb source operand from the GPR.
15	GPR Write	When this bit is set, this instruction writes a result to the Rt register in the GPR.

Table 12-7: **Bit Descriptions for the UDI Configuration Register *(Cont'd)***

Bit	Name	Description
16	CR Enable	When this bit is set, this instruction returns Condition Record (CR) bits to the CR field indicated in CRField[0:2].
17	Reserved	Reserved
18:20	CRField[0:2]	Indicates which field receives the condition record (if the CR Enable bit in the instruction is set to 1).
21	Reserved	Reserved
22:25	Register #	Indicates the register number (0 – 15). This value is only used when setting the DCR read/write pointer (Type = `2'b11`), but can be viewed when reading the UDI contents through a DCR read.
26:27	Type	Indicates the operation type of the instruction: • `00`: Non-autonomous, early confirm • `01`: Non-autonomous, late confirm • `10`: Autonomous • `11`: Sets read/write pointer to value in Register # field
28:29	Reserved	Reserved
30	Wildcard	When this bit is set, bits [1:5] are not considered as the extended opcode but can be anything. Instead only bits [6:11] are checked. When this bit is cleared, the entire 11 bits of the extended opcode are checked.
31	En	This bit enables the UDI. It indicates that the opcode and options written in the UDI are valid and should be used during decode.

Clocking

The FCM can be clocked at integer multiples of the processor clock. The clock ratio between the processor and FCM can range from 1:1 up to 16:1. In other words, the FCM can run at the same speed as the processor or slower. The two clocks must be rising-edge aligned.

The APU controller uses the processor clock for the APU controller/processor interface as well as its internal logic. All inputs and outputs on the APU controller/FCM interface are synchronized using the FCM clock.

Processor Migration

This section describes key points for customers migrating their APU hardware and software IP from the 405 processor to the 440 processor.

New Features

To improve performance, the APU controller supports a 128-bit load bus and a 128-bit store bus. The transfer of a full quadword can occur in one FCM clock cycle. Previously, this transfer took four clock cycles.

A new signal, APUFCMNEXTINSTRREADY, helps to increase instruction throughput. This signal is driven High when the next load instruction and all of its data are ready to be sent to the FCM. A High on this signal means that as soon as FCMAPUDONE is asserted for the current instruction, the APU controllers sends the next load instruction with all data on the next FCM cycle. Because the FCM knows when the next instruction will arrive, it can drive FCMAPUDONE High during that same cycle, which allows for the load instruction to be sent and completed all in one FCM clock cycle.

New "wildcard" option for each of the 16 UDI registers.

The user must specify the CR enable option in the UDI register for any instruction that returns CR status. Previously, this bit was not needed.

APU Controller now pipelines up to three instructions internally, making it possible to achieve back-to-back instructions on the FCM interface.

APUFCMMSRFE0 and APUFCMMSRFE1 are new signals for use with the FPU. These signals determine the precise/imprecise exception mode for floating-point instructions.

FPU exceptions have OS support automatically, and no special APU configuration is needed.

FCMAPUFPSCRFEX is a new signal used for FPU generated exceptions. It reflects the value of FPSCR[FEX].

The APU Controller now provides decoded information to the FCM that indicates if the instruction is a load or a store, its transfer size, and if it is an FPU operation or a non-autonomous instruction.

During an FCM transaction, the signals from the APU remain valid until the end of the transaction. The only signals that are pulsed are APUFCMWRITEBACKOK and APUFCMFLUSH.

The new FCMAPUCONFIRMINSTR signal is used for non-autonomous instructions with late confirmation to indicate that the FCM will not generate an exception for that instruction.

Dropped Features

UDIs no longer send Carry or Overflow information back to the processor. The embedded processor does not support Carry or Overflow reporting for UDIs. However, UDIs can still send their own Condition Record (CR) bits to any of eight CR fields and are defined by the user.

Storage UDIs (loads and stores) no longer force big-Endian steering. The embedded processor in Virtex-5 FPGAs does not support this feature. Instead the FCM needs to watch the Endian bit of the hardware and take care of any byte swapping based on its own Endian configuration.

In the previous version of the APU controller, the FCM could decode and "steal" the Integer Divide instruction from the processor and execute it in the FCM. This feature is not supported by the embedded processor in Virtex-5 FPGAs.

FCM decoded instructions are no longer supported. Previously, the APU Controller passed an unknown instruction to the FCM for the FCM to decode. This is no longer possible. Only instructions decoded by the APU Controller are passed to the FCM.

Non-autonomous blocking instructions have been replaced with non-autonomous with early confirmation instructions. This new class of instructions is similar to non-autonomous blocking instructions; however, the APUFCMWRITEBACKOK signal can no longer be received in a specific cycle.

Non-autonomous non-blocking instructions have been replaced with non-autonomous with late confirmation instructions. This new class is similar to non-autonomous non-blocking instructions; however, the APUFCMWRITEBACKOK signal is received before FCMAPUDONE is asserted in response.

Interface Changes

This section describes the changes in the APU to FCM interface.

Table 12-8: **FCM Interface Signals**

FCM Interface Signal	Direction	Function
APUFCMDECFPUOP(1)	Output	APU controller decoded FPU instruction.
APUFCMDECLDSTXFERSIZE[0:2](1)	Output	APU controller decoded load/store transfer size. • `100`: byte • `010`: halfword • `001`: word • `011`: doubleword • `111`: quadword
APUFCMDECLOAD(1)	Output	APU controller decoded load instruction.
APUFCMDECNONAUTON(1)	Output	APU controller decoded instruction that is a non-autonomous instruction. Store instructions are also included because stores return data to the CPU.
APUFCMDECSTORE(1)	Output	APU controller decoded store instruction.
APUFCMDECUDI[0:3](2)	Output	Specifies the UDI register decoded.
APUFCMINSTRVALID(2)	Output	Indicates if the instruction on APUFCMINSTRUCTION[0:31] is valid. This signal now only applies to instructions decoded by the APU controller.
APUFCMLOADBYTEADDR[0:3](2)	Output	Now specifies at which of the 16 bytes the data begins.
APUFCMLOADDATA[0:127](2)	Output	128-bit load data bus. This bus is now wider.
APUFCMLOADDVALID(2)	Output	APUFCMLOADDATA[0:127] is valid.
APUFCMMSRFE0(1)	Output	This signal indicates the value of MSR[FE0]. It is used for FPU instructions only.
APUFCMMSRFE1(1)	Output	This signal indicates the value of MSR[FE1]. It is used for FPU instructions only.

Table 12-8: **FCM Interface Signals *(Cont'd)***

FCM Interface Signal	Direction	Function
APUFCMNEXTINSTRREADY[(1)]	Output	This signal indicates the APU controller sends the next instruction, along with all data, on the clock cycle after the current FCM instruction is finished (when FCMAPUDONE is received).
FCMAPUCONFIRMINSTR[(1)]	Input	Indicates the FCM does not cause an exception for this instruction. This signal is used for non-autonomous with late confirmation instructions.
FCMAPUFPSCRFEX[(1)]	Input	Indicates an FPU instruction generated an exception. This should be the value of FPSCR[FEX] bit in the FPU.
FCMAPUSTOREDATA[0:127][(1)]	Input	128-bit separate store data bus.

Notes:

1. This is a new signal or bus.
2. This signal has changed in bus size or function from the 405 version of the signal.

Table 12-9 summarizes signals that were in the previous APU controller/FCM interface but are no longer available.

Table 12-9: **Unsupported Signals**

Signal	Function
FCMAPUDCDXEROVEN	FCM decoded instruction that returns overflow.
FCMAPUDCDXERCAEN	FCM decoded instruction that returns carry.
FCMAPUDCDFORCEBESTEERING	FCM decoded load/store that forces big-Endian steering.
FCMAPUXEROV	Overflow result of FCM instruction.
FCMAPUXERCA	Carry result of FCM instruction.
FCMAPUINSTRACK	Valid instruction decoded in the FCM.
FCMAPUDECODEBUSY	Allows FCM to do a multicycle instruction decode before returning FCMPAUINSTRACK.
FCMAPUDCDGPRWRITE	FCM decoded instruction writes back to GPR.
FCMAPUDCDRAEN	FCM decoded instruction needs data from GPR(Ra).
FCMAPUDCDRBEN	FCM decoded instruction needs data from GPR(Rb).
FCMAPUDCDPRIVOP	FCM decoded instruction executes in privileged mode.
FCMAPUDCDFORCEALIGN	FCM decoded load/store instruction with forced alignment.
FCMAPUDCDCREN	FCM decoded instruction sets condition record (CR) bits.
FCMAPUEXECRFIELD[0:2]	FCM decoded instruction sets which CR field to update.
FCMAPUDCDLOAD	FCM decoded load instruction.

Table 12-9: **Unsupported Signals *(Cont'd)***

Signal	Function
FCMAPUDCDSTORE	FCM decoded store instruction.
FCMAPUDCDUPDATE	FCM decoded load/store with update.
FCMAPUDCDLDSTBYTE	FCM decoded load/store byte transfer.
FCMAPUDCDLDSTHW	FCM decoded load/store halfword transfer.
FCMAPUDCDLDSTWD	FCM decoded load/store word transfer.
FCMAPUDCDLDSTDW	FCM decoded load/store doubleword transfer.
FCMAPUDCDLDSTQW	FCM decoded load/store quadword transfer.
FCMAPUDCDTRAPLE	FCM decoded load/store that causes an alignment exception if the Endian attribute is `1'b1`.
FCMAPUDCDTRAPBE	FCM decoded load/store that causes an alignment exception if the Endian attribute is `1'b0`.
FCMAPUFPUOP	FCM decoded FPU instruction.
FCMAPUEXEBLOCKINGMCO	FCM decoded instruction of blocking class.
FCMAPUEXENONBLOCKINGMCO	FCM decoded instruction of blocking class.
FCMAPULOADWAIT	FCM is not yet ready to receive next load data. This signal is no longer necessary because data remains valid until FCMAPUDONE is asserted.
APUFCMDECODED	Indicates the APU controller decoded the instruction. Because there are now only APU controller decoded instructions, this signal is not necessary.
APUFCMXERCA	Carry in for extended arithmetic.

Timing Diagrams for the APU Controller

This section provides timing diagrams that show the maximum throughput of instructions. The examples show waveforms for autonomous instructions, quadword loads, non-autonomous instructions with early confirm, quadword stores, non-autonomous instructions with late confirm, APU enabled exceptions (both accepted by CPU and flushed), and FPU enabled exceptions (both accepted by CPU and flushed). For each transaction, the instruction, decoded information, and any source or load data remains valid until the APU controller receives an asserted FCMAPUDONE. APUFCMWRITEBACKOK or APUFCMFLUSH are pulsed once for each instruction. All signals coming from the FCM should be pulsed for one FCM clock signal unless stated otherwise.

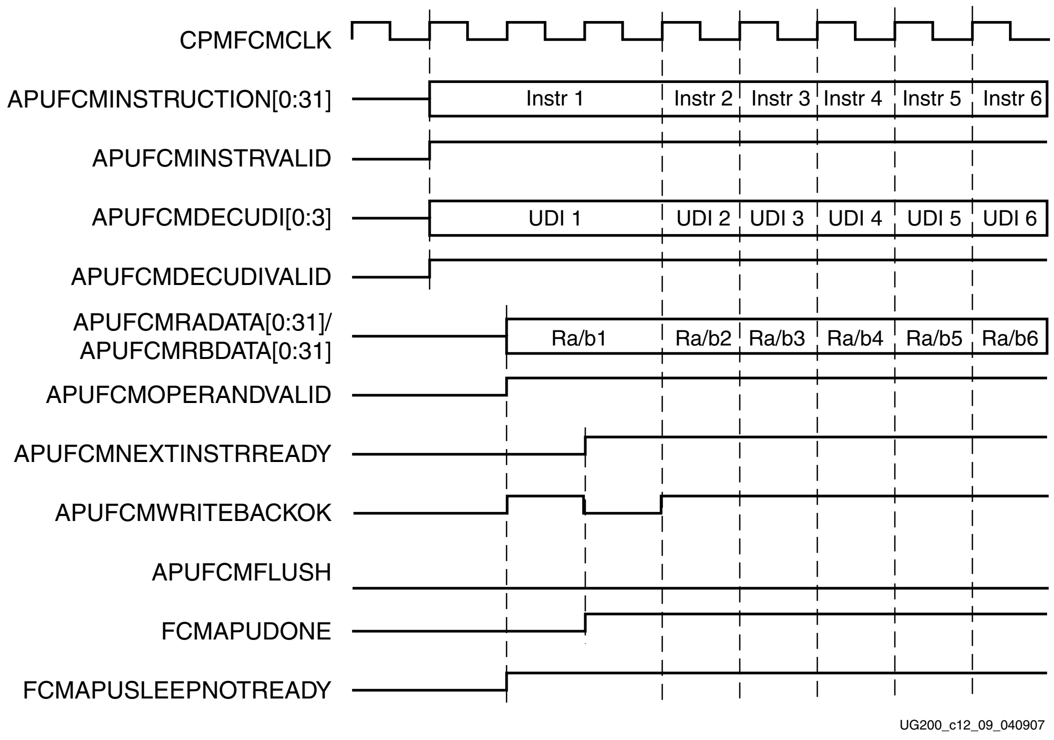


Figure 12-9: **Autonomous Back-to-Back Instructions (3:1 Clock Ratio or Higher)**

Figure 12-9 shows back-to-back autonomous instructions. To achieve the back-to-back instructions on the FCM interface, the clock ratio must be 3:1 or larger. This timing diagram assumes the FCM signals are flopped (the FCM cannot respond in the same clock cycle that an instruction is sent). (Refer to Figure 12-10 for an example with an FCM that uses the same cycle response logic.) If the FCM can respond using same cycle logic, the APU controller can achieve back-to-back transactions at a 2:1 clock ratio. The first instruction is sent to the FCM as soon as possible along with the UDI number that was decoded. The operands along with APUFCMWRITEBACKOK are ready in the next cycle. The FCM can then respond by asserting FCMAPUDONE on the next clock cycle. In the meantime, because the APU controller can accept up to three instructions, the APU controller has received the next autonomous instruction and its operands. Because the APU controller has all pieces of the second UDI, it asserts APUFCMNEXTINSTRREADY High, so that the cycle after FCMAPUDONE is asserted, the next instruction will be presented with its associated data. The FCM then can assert FCMAPUDONE on the following clock cycle. At this point, new instructions can be sent to the FCM every clock cycle.

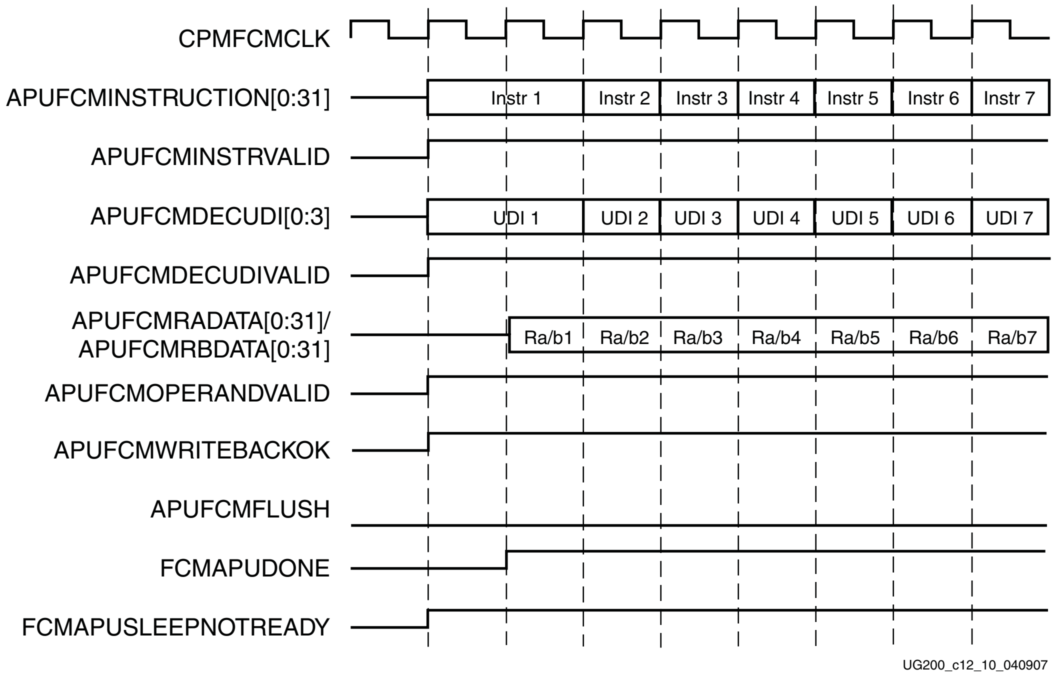


Figure 12-10: **Autonomous Back-to-Back Instructions (2:1 Clock Ratio or Higher, Assumes Same Cycle Response)**

Figure 12-10 shows back-to-back autonomous instructions. To achieve the back-to-back instructions on the FCM interface (assuming the FCM uses combinatorial logic), the clock ratio must be 2:1 or larger. Figure 12-10 assumes the FCM signals are not flopped (the FCM can respond in the same clock cycle an instruction is sent). In this diagram, the first instruction is sent to the FCM as soon as possible along with the decoded UDI number. The operands are ready in the next cycle along with APUFCMWRITEBACKOK. The FCM can then respond with FCMAPUDONE on the same clock cycle. In the meantime, because the APU Controller can accept up to three instructions, the APU Controller has received the next Autonomous instruction along with its operands. At this point, new instructions can be sent to the FCM every clock cycle.

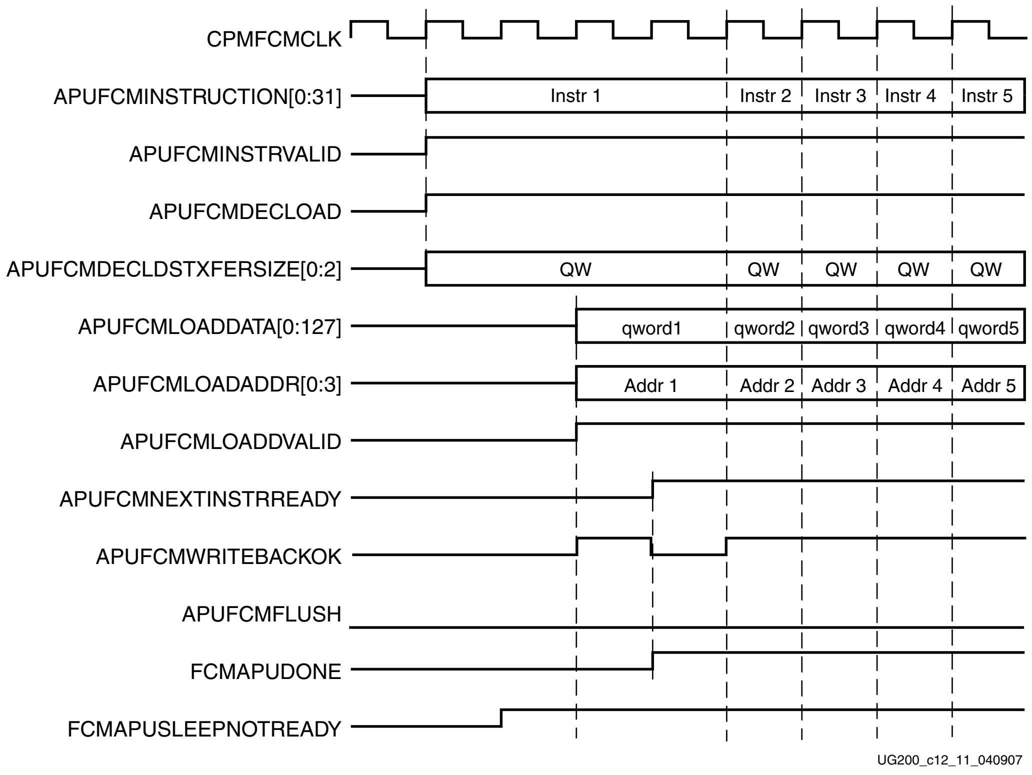


Figure 12-11: **Quadword Load Back-to-Back Instructions (4:1 Clock Ratio or Higher)**

Figure 12-11 shows back-to-back quadword loads. At clock ratios of 4:1 and larger, the FCM can receive one quadword each cycle. At smaller clock frequencies, the FCM can receive two quadwords every three cycles with more bubbles at a 1:1 clock ratio. Figure 12-11 assumes the FCM signals are flopped. (Refer to Figure 12-12 for an example of a quadword load that uses the same cycle response logic in the FCM.) If the FCM can achieve same cycle response, back-to-back transfers are possible each FCM clock cycle at a 3:1 clock ratio. In this timing diagram, the instruction is sent immediately with APUFCMDECLOAD and APUFCMDECLDSTXFERSIZE[0:2]. When the load data has been sent by the processor, the APU controller passes the load data to the FCM with APUFCMLOADADDR[0:3], which indicates the byte at which the data begins on the load data bus. By this time, the second instruction and its associated data have been received. When the FCM asserts FCMAPUDONE for instruction 1, the APU controller asserts APUFCMNEXTINSTRREADY High to indicate it will send all the data for the second instruction on the next cycle. The FCM can then assert FCMAPUDONE High for instruction 2. This pattern continues for back-to-back quadword loads so that the FCM receives one load every clock cycle.

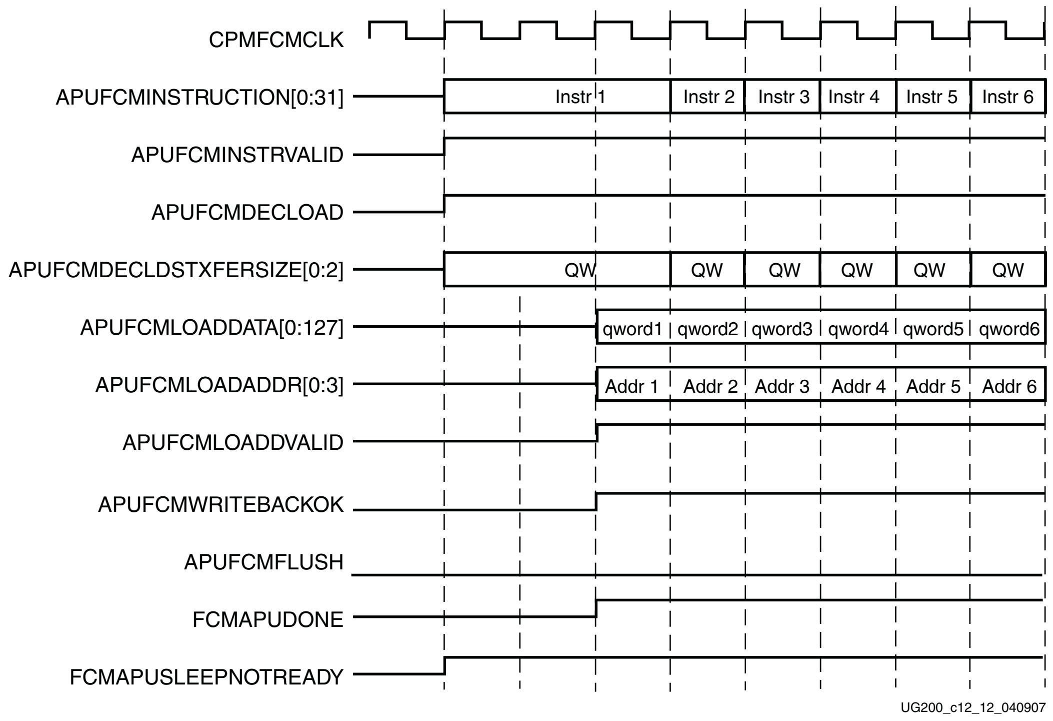


Figure 12-12: **Quadword Load Back-to-Back Instructions (3:1 Clock Ratio or Higher)**

Figure 12-12 shows back-to-back quadword loads. At clock ratios of 3:1 and larger, the FCM can receive one quadword each cycle. Figure 12-12 assumes the FCM signals are not flopped. In this timing diagram, the instruction is sent immediately along with APUFCMDECLOAD and APUFCMDECLDSTXFERSIZE[0:2]. Once the load data has been sent by the processor, the APU Controller passes the load data to the FCM along with APUFCMLOADADDR[0:3], which indicate the byte at which the data begins on the load data bus. The FCM can then respond with FCMAPUDONE in the same cycle. This pattern continues for back-to-back quadword loads so that the FCM can receive one every clock cycle.

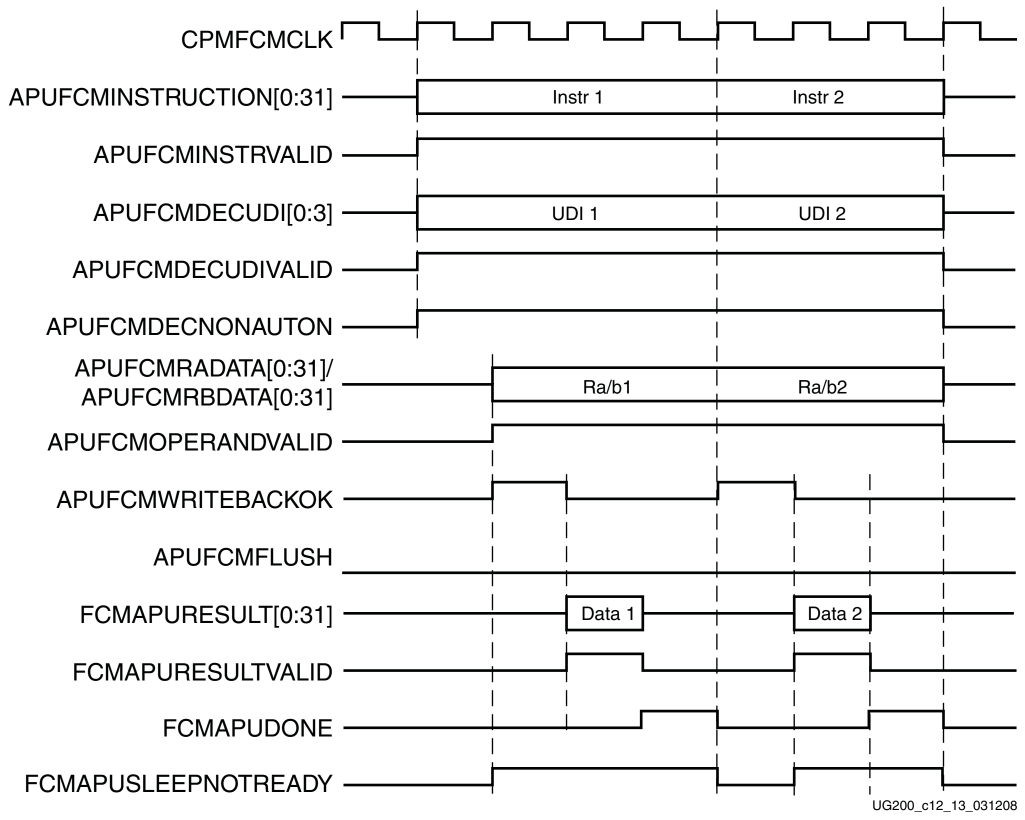


Figure 12-13: **Non-Autonomous Instructions with Early Confirm Back-to-Back (3:1 Clock Ratio or Higher)**

Figure 12-13 shows back-to-back non-autonomous instructions with early confirm. At clock ratios of 3:1 or larger, the FCM can return a result every other clock cycle. Figure 12-13 assumes the FCM signals are flopped. (Refer to Figure 12-14 for an example of an FCM with combinatorial logic.) In Figure 12-13, the first instruction is sent to the FCM as soon as possible along with the decoded UDI number and APUFCMDECNONAUTON, which indicates the instruction type (non-autonomous). The operands are ready with APUFCMWRITEBACKOK in the next cycle. The FCM can then respond with the result data (FCMAPURESULT[0:31], FCMAPURESULTVALID) on the next clock cycle. In the subsequent clock cycle, the FCMAPUDONE is asserted. The next instruction has all of its source data, so both the instruction and data can be sent in the same cycle. The FCM then returns its result in the following cycle, allowing for one FCM instruction every three clock cycles.

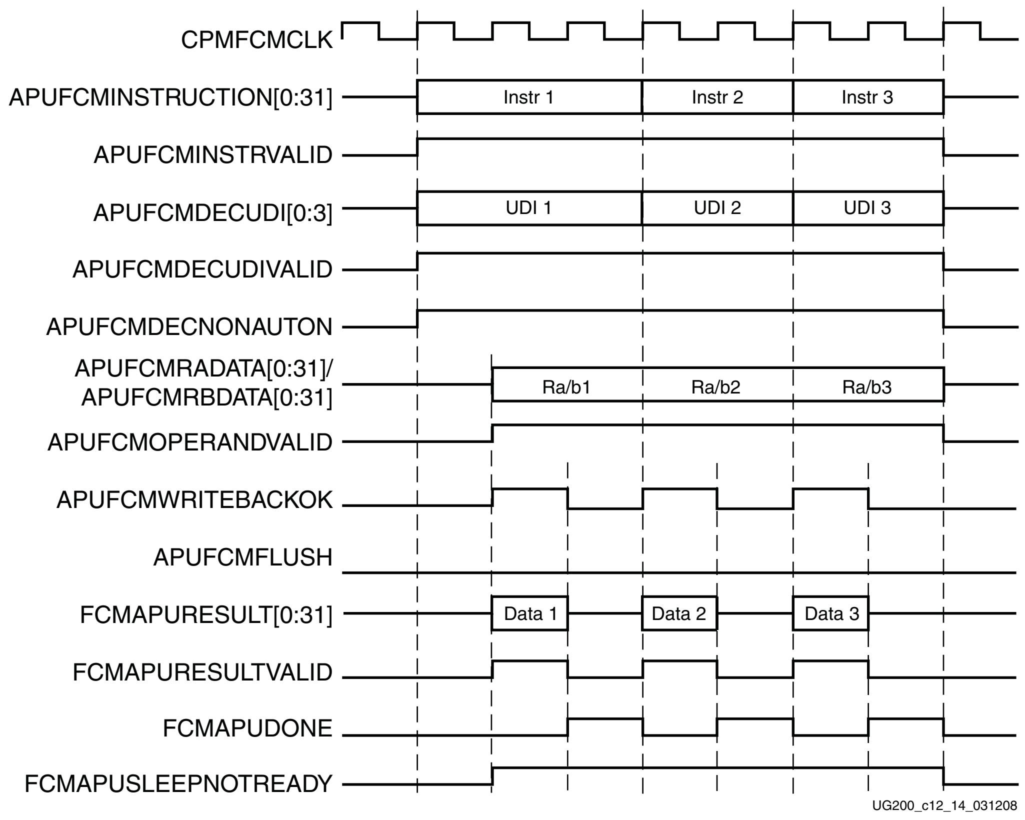


Figure 12-14: **Non-Autonomous Instructions with Early Confirm Back-to-Back (2:1 Clock Ratio or Higher)**

Figure 12-14 shows back-to-back non-autonomous instructions with early confirm. At clock ratios of 2:1 or larger, the FCM can return a result every other clock cycle. Figure 12-14 assumes the FCM signals are not flopped. In this diagram, the first instruction is sent to the FCM as soon as possible along with the decoded UDI number and APUFCMDECNONAUTON, indicating that the instruction is of type Non-autonomous. The operands are ready in the next cycle along with APUFCMWRITEBACKOK. The FCM can then respond with the result data (FCMAPURESULT[0:31], FCMAPURESULTVALID) in the same clock cycle. Because the next instruction has all of its source data, both the instruction and data can be sent in the same cycle. The FCM then returns its result immediately and in the next clock cycle asserts FCMAPUDONE, allowing for one FCM instruction every two clock cycles.

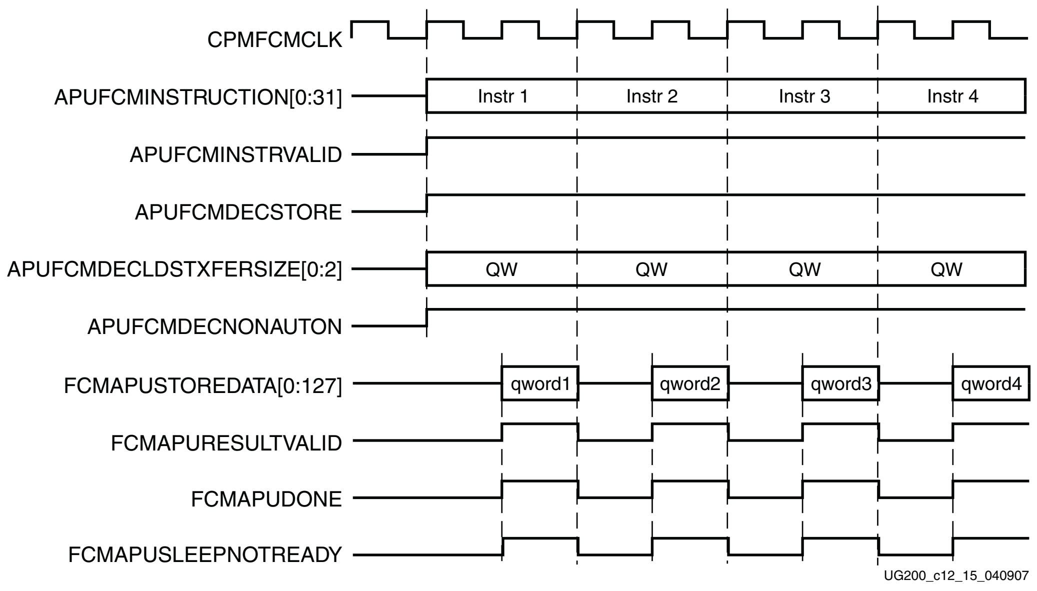


Figure 12-15: **Quadword Store Back-to-Back Instructions without WritebackOK (2:1 Clock Ratio or Higher)**

Figure 12-15 shows back-to-back quadword stores. At clock ratios of 2:1 and larger, the FCM can receive one quadword every other cycle. Figure 12-15 assumes the FCM signals are flopped. The FCM signals must be combinatorial in order to send a quadword store each clock cycle. (Refer to Figure 12-16 for an example of a combinatorial response from the FCM.) In Figure 12-15, the instruction is sent immediately along with APUFCMDECSTORE, APUFCMDECLDSTXFERSIZE[0:2], and APUFCMDECNONAUTON. Any instruction that returns data or status to the CPU is considered to be non-autonomous. A store acts as a non-autonomous instruction with early confirm. Because the store in this example is without WritebackOK, the FCM can then respond with FCMAPUSTOREDATA[0:127], FCMAPURESULTVALID, and FCMAPUDONE on the next FCM clock. On the next cycle, the APU controller can send the second store, allowing the FCM to send one quadword store every other clock cycle.

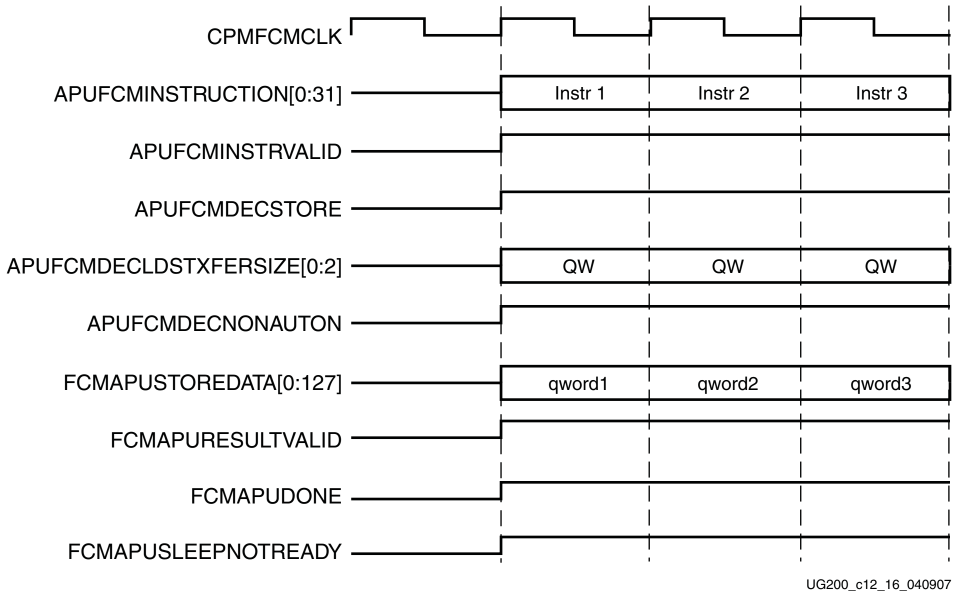


Figure 12-16: **Quadword Store Back-to-Back Instructions without WritebackOK (2:1 Clock Ratio or Higher) and with Same Cycle Response from FCM**

Figure 12-16 shows back-to-back quadword stores. At clock ratios of 2:1 and larger, the FCM can receive one quadword every clock cycle. Figure 12-16 assumes the FCM signals are not flopped. The FCM signals must be combinatorial in order to send a quadword store each clock cycle. In this timing diagram, the instruction is sent immediately along with APUFCMDECSTORE, APUFCMDECLDSTXFERSIZE[0:2], and APUFCMDECNONAUTON. Any instruction that returns data or status to the CPU is considered to be Non-autonomous. A store instruction acts as a non-autonomous instruction with early confirm. Because the store in this example is without WritebackOK, the FCM can respond with FCMAPUSTOREDATA[0:127], FCMAPURESULTVALID, and FCMAPUDONE on the same FCM clock. On the next cycle, the APU controller can send the second store, allowing the FCM to send one quadword store every clock cycle.

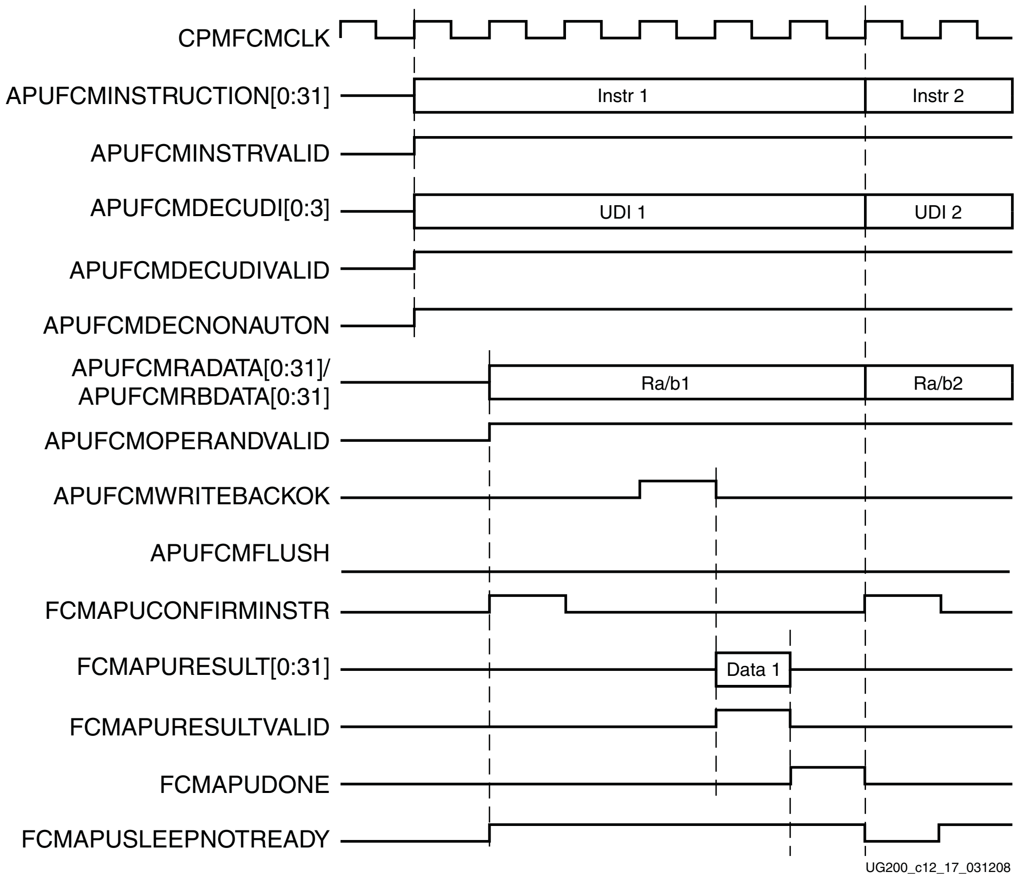


Figure 12-17: **Non-Autonomous Instructions with Late Confirm Back-to-Back (2:1 Clock Ratio or Higher)**

Figure 12-17 shows back-to-back non-autonomous instructions with late confirm. At clock ratios of 2:1 or larger, the FCM can return a result every six clock cycles. Figure 12-17 assumes the FCM signals are flopped. In Figure 12-17, the first instruction is sent to the FCM as soon as possible along with the decoded UDI number and APUFCMDECNONAUTON, which indicates the instruction type (non-autonomous). The operands are ready in the next cycle, and at the same time the FCM asserts FCMAPUCONFIRMINSTR, indicating that the instruction will not cause a precise exception. When the APU controller has received FCMAPUCONFIRMINSTR, it asserts APUFCMWRITEBACKOK as soon as possible. APUFCMWRITEBACKOK is asserted at the earliest cycle in this example. The FCM can then respond with the result data (FCMAPURESULT[0:31], FCMAPURESULTVALID) on the next clock cycle. The next instruction has all of its source data, so it can send the instruction and data in the same cycle. The FCM then confirms the instruction will not cause an exception.

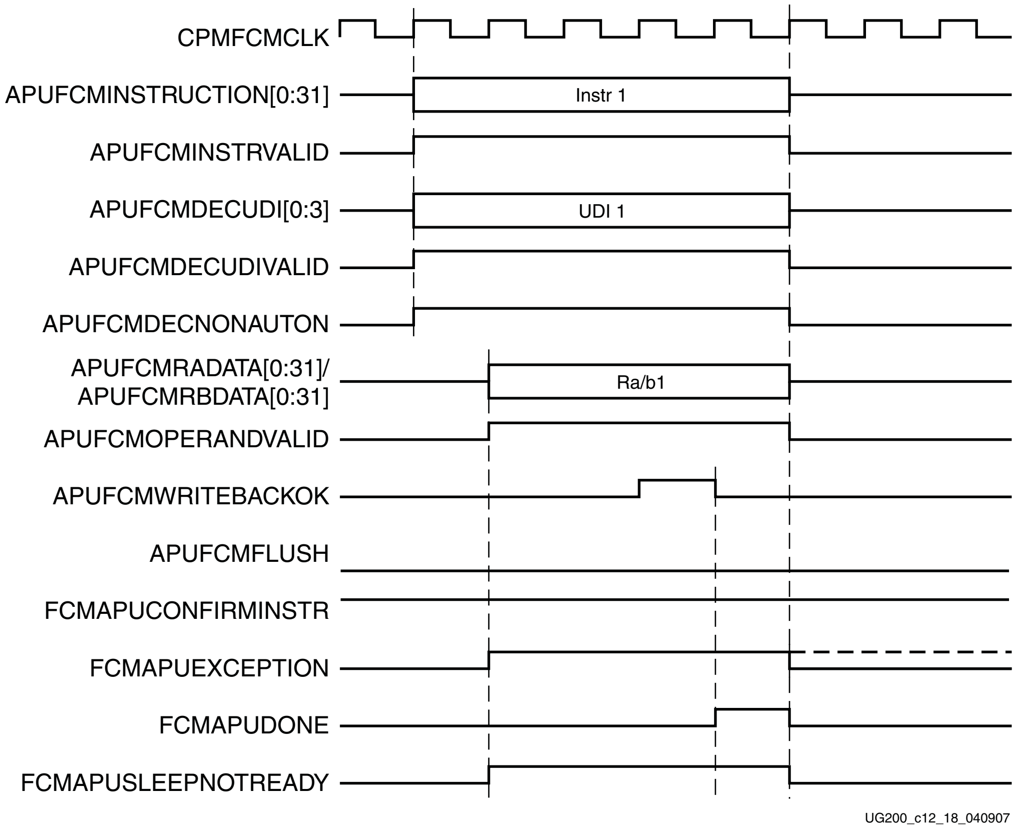


Figure 12-18: **APU Enabled Exception that is Received by the CPU**

Figure 12-18 shows an APU exception (generated by the FCM). Figure 12-18 assumes the FCM signals are flopped. The instruction must be non-autonomous with late confirm. Instead of asserting FCMAPUCONFIRMINSTR after receiving the instruction, the FCM holds FCMAPUEXCEPTION High. This signal can be driven anytime after the FCM has received the instruction; it does not have to wait for operand data. In this case, the CPU takes the APU enabled exception; no other interrupt has higher priority. The APU controller asserts APUFCMWRITEBACKOK to let the FCM know the exception was received. During that cycle or at a later cycle, the FCM must assert FCMAPUDONE to complete the transaction. When FCMAPUDONE is asserted, the FCM can deassert FCMAPUEXCEPTION or, if wanted, leave it High to be cleared by software. If the FCM keeps FCMAPUEXCEPTION High, any later instructions of type non-autonomous with late confirm are seen as causing an exception.

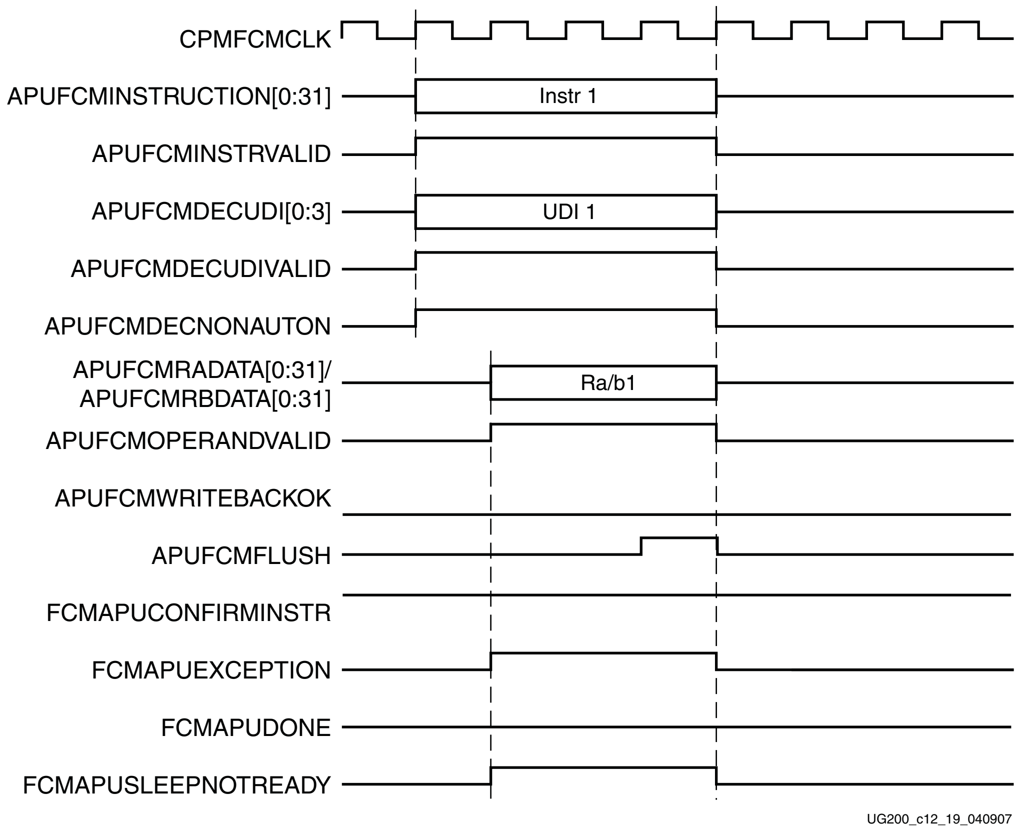


Figure 12-19: **APU Enabled Exception that is Flushed by the CPU**

Figure 12-19 shows an APU exception (generated by the FCM). Figure 12-19 assumes the FCM signals are flopped. The instruction must be non-autonomous with late confirm. Instead of asserting FCMAPUCONFIRMINSTR after receiving the instruction, the FCM holds FCMAPUEXCEPTION High. This signal can be asserted anytime after the FCM has received the instruction; it does not have to wait for operand data. In this case, the CPU takes an earlier or higher priority interrupt and flushes the APU instruction that is trying to generate an exception. The APU controller asserts APUFCMFLUSH to let the FCM know the exception was flushed. When APUFCMFLUSH is asserted, the FCM must deassert FCMAPUEXCEPTION. The FCM should not assert FCMAPUDONE.

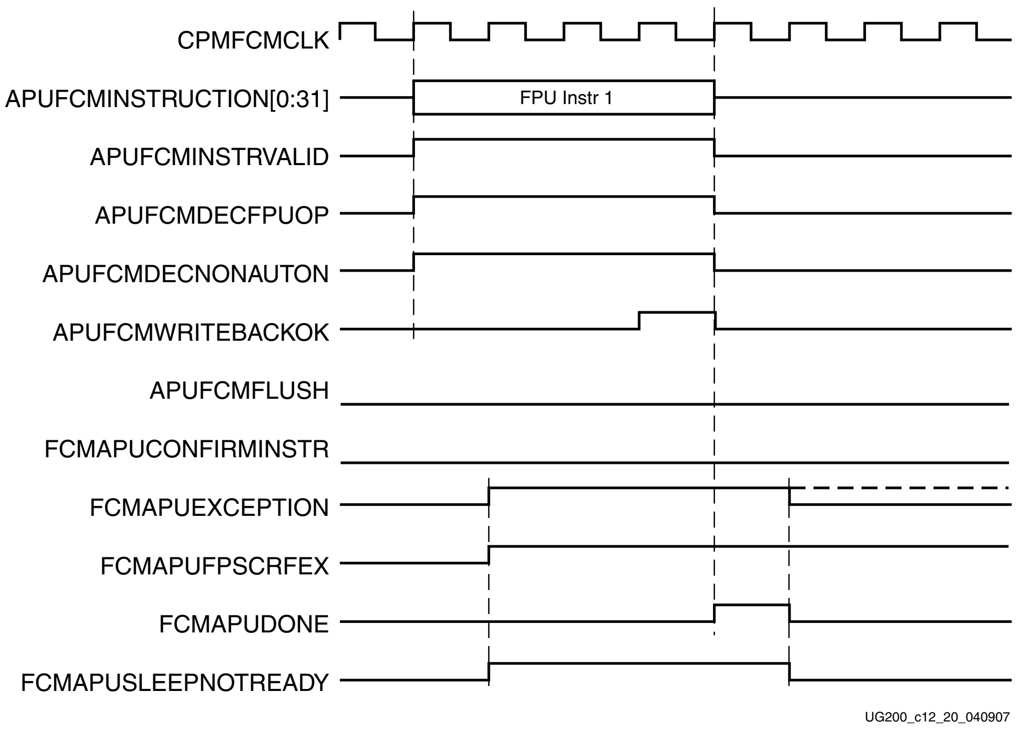


Figure 12-20: **FPU Enabled Exception that is Received by the CPU**

Figure 12-20 shows an FPU exception (generated by the FCM). Figure 12-20 assumes the FCM signals are flopped. The instruction must be non-autonomous with late confirm and must set the FPSCR[FEX] bit. Instead of asserting FCMAPUCONFIRMINSTR after receiving the instruction, the FCM holds FCMAPUEXCEPTION and FCMAPUFPSCRFEX High. These signals can be asserted anytime after the FCM has received the instruction. In this case, the CPU takes the FPU enabled exception; no other interrupt has higher priority. The APU controller asserts APUFCMWRITEBACKOK to let the FCM know the exception was received. During that cycle or at a later cycle, the FCM must assert FCMAPUDONE to complete the transaction. When FCMAPUDONE is asserted, the FCM can deassert FCMAPUEXCEPTION or, if wanted, this signal can remain High to be cleared by software. If the FCM keeps FCMAPUEXCEPTION High, any later instructions of type non-autonomous with late confirm are seen as causing an exception. FCMAPUFPSCRFEX must remain High until cleared by software. For the CPU to recognize an FPU exception, MSR[FE0,FE1] must be non-zero. Otherwise the exception is not seen by the CPU until these bits become non-zero.

Chapter 13

DMA Controller

The DMA controller consists of four independent DMA engines that provide high-performance direct memory access for streaming data. Peripherals can directly transfer data to and from a memory controller connected to the processor block. Peripherals are connected to the DMA engines through the LocalLink interface. The DMA engines can be monitored and controlled through their Device Control Registers (DCRs).

DMA Controller Features

The key features of the DMA controller are listed below:

- Four complete full-duplex DMA engines
- Generic LocalLink interfaces stream data in and out of the engines
- Efficient command translation generates bursts from and to memory for payload transfers
- 32-byte aligned bursts for DMA descriptor reads
- Efficient 16-byte aligned commands for DMA descriptor writes
- High-performance payload read pipelining to the target memory
- Non-blocking RX and TX operations with respect to the target memory
- Asynchronous LocalLink clock allows the user logic to run at any frequency relative to the processor block
- DMA engines broadcast application-specific data across the LocalLink interfaces
- Separate DMA engine reset feature for locked-up engines
- Dynamic descriptor appending
- Interrupt coalescing mechanism
- Interrupt delay timer mechanism
- Simple software use model

DMA Operation

Each DMA engine consists of two independent DMA channels, one for TX and the other for RX, allowing full-duplex operation per engine. Figure 13-1 shows a high-level block diagram of a single DMA engine.

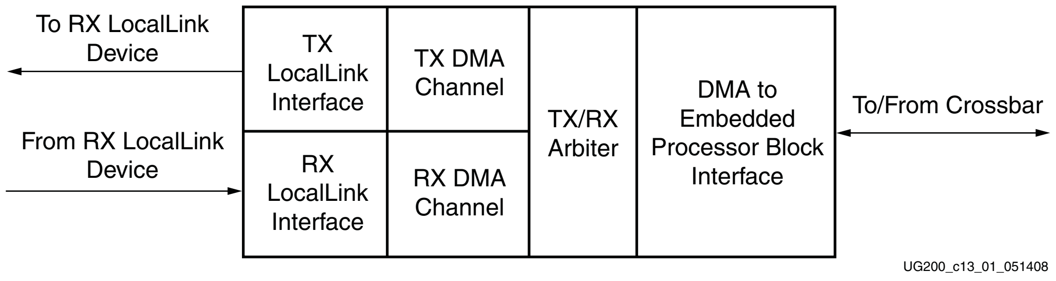


Figure 13-1: **High-Level Block Diagram of Single DMA Engine**

The LocalLink protocol is used to transfer data packets from peripherals to memory, and from memory to peripherals. A LocalLink packet consists of the packet header, the packet payload, and the packet footer. The DMA TX engine collects data from one or more contiguous regions of memory to create the payload for a data packet that it then transmits over the TX LocalLink interface. The DMA RX engine receives a data packet from a peripheral, and writes the payload to one or more memory locations as specified by the control registers (descriptors).

Each DMA channel is controlled by *separate* descriptors, which are data structures set up by the CPU before the DMA operations commence. Among other things, these descriptors control how much data is to be transferred and the location of the data in system memory. Descriptors can be chained together, allowing either a sequence of separate memory blocks to be combined into a transmitted data packet or a received data packet to be broken up and saved in a sequence of separate memory blocks.

The CPU sets up the DMA by first creating the sequence of descriptors in memory and then writing the address of the first descriptor to the Current Descriptor Pointer DCR register. Finally, the CPU starts the DMA operation by writing the address of the last descriptor in the sequence to the Tail Descriptor Pointer DCR register.

This last write action triggers the DMA engine to fetch a new descriptor from the location pointed to by the Current Descriptor Pointer register. In the case of a transmit channel, the DMA engine starts fetching data from the memory locations indicated by the descriptor, and starts the process of creating and sending a data packet. After transmitting all the data indicated by the current descriptor, the DMA engine fetches the next descriptor, if any, and continues to transmit data indicated by that descriptor.

In the case of a receive channel, the DMA engine waits for the data packet to be received from the external peripheral, and starts copying the received data to the memory locations indicated by the current descriptor. If more data is received, the next descriptor is fetched, and the received data is copied to the corresponding memory locations. This process continues until the end of the received payload.

Descriptor Format

The descriptor consists of eight words as shown in Table 13-1.

Table 13-1: **Descriptor Format**

Word#	Byte Offset	Descriptor Field	
		MSB	LSB
0	`0x00`	Next Descriptor Pointer	
1	`0x04`	Buffer Address	
2	`0x08`	Buffer Length	
3	`0x0C`	Sts/Ctrl	Application-Defined Data
4	`0x10`	Application-Defined Data	
5	`0x14`	Application-Defined Data	
6	`0x18`	Application-Defined Data	
7	`0x1C`	Application-Defined Data	

The Next Descriptor Pointer field indicates from where in memory the next descriptor should be fetched. This field must be eight-word aligned (the five least-significant bits must be 0s). The Buffer Address field is a byte aligned address pointing to the payload source/destination. The Buffer Length field is the length of the payload to transfer in bytes. The Sts/Ctrl field is a single byte that contains status and control information for the DMA channel. The Application Defined Data fields are for the explicit use of the application and are broadcasted over the LocalLink interface at appropriate times.

For the TX channel, the application data is transmitted down the LocalLink interface during the first descriptor that sets the Start of Packet (SOP) bit on the LocalLink interface. For the RX channel, the application data is received from the LocalLink interface and written back to the last DMA descriptor that was in progress when the LocalLink interface encountered an End of Packet (EOP). Details of how descriptor information is transferred to and from a LocalLink packet are provided in "DMA TX LocalLink Interface," page 231 and "DMA RX LocalLink Interface," page 233. See [Ref 6] for more information on the LocalLink interface.

The Sts/Ctrl byte format is shown in Table 13-2.

Table 13-2: **Descriptor Status/Control Byte Format**

Bit#	Sts/Ctrl Field	Field Type	Description
0 (msb)	DMA_ERROR	Status	The DMA sets this bit when an error is encountered. It is a copy of the Error Interrupt bit status (see "Interrupt Mechanism," page 236).
1	DMA_INT_ON_END	Ctrl	The CPU sets this bit to cause the DMA to generate an interrupt event when the current descriptor has been completed.
2	DMA_STOP_ON_END	Ctrl	The CPU sets this bit to cause the DMA channel to halt when the current descriptor has been completed. The DMA can be restarted by rewriting to the Tail Descriptor Pointer register. DMA_STOP_ON_END and DMA_INT_ON_END are independent of each other. As such, the DMA can be made to do any of four possible operations: • Halt with an interrupt • Halt without an interrupt • Interrupt without halting • Nothing at all An alternate mechanism for halting the channel is when the descriptor Tail Pointer equals the descriptor Current Pointer.
3	DMA_COMPLETED	Status	The DMA sets this bit to indicate that the current descriptor has been completed (the payload is transferred). • For the TX Channel: Set when the Buffer Length decrements to zero. • For the RX Channel: Set when the Buffer Length decrements to zero or when EOP is received on the RX LocalLink interface. The Buffer Length does not specify how much data was transferred in this case.
4	DMA_START_OF_PACKET	Status/Ctrl	• TX Channel: (Ctrl) The CPU sets this bit to instruct the LocalLink interface to initiate a header for the packet. • RX Channel: (Status) When an SOP is asserted on the LocalLink RX interface, the DMA sets this bit in the descriptor.
5	DMA_END_OF_PACKET	Status/Ctrl	• TX Channel: (Ctrl) The CPU sets this bit to instruct the LocalLink interface to initiate a footer for the packet. • RX Channel: (Status) When an EOP is asserted on the LocalLink RX interface, the DMA sets this bit in the descriptor.
6	DMA_CHANNEL_BUSY	Status	The DMA sets this bit to indicate that the DMA Channel is busy. No DMA registers should be written during this time (except for the Descriptor Tail Pointer). Register reads are allowed at any time.
7 (lsb)	Undefined	N/A	N/A

Using Descriptors to Describe a Packet

The descriptors can be used in two ways to describe a packet:

1. A single descriptor describes a packet in its entirety.
2. Multiple descriptors are chained together to describe a single packet.

For the first case, the SOP and EOP flags are both set in the same descriptor. For TX, these are set by the CPU. For RX, they are set by the DMA when the LocalLink interface receives them.

In the second case, multiple non-contiguous descriptors are chained together to form an apparently contiguous data payload across the LocalLink interface.

DMA Legacy Mode

A legacy mode is supported for users who want to port legacy designs using older Virtex devices. In this legacy mode, a write to the DCR Current Descriptor Pointer register triggers a DMA operation, and the DCR Tail Descriptor Pointer register is not used. This mode is enabled by writing a 0 to the TailPtrEn field of the control register. When this legacy mode is enabled, the Tail == Current Pointer comparison is not used. Thus the Dynamic Descriptor Appending Mechanism cannot be used in this legacy mode (see "Dynamic Descriptor Appending," page 239) because the Tail Descriptor Pointer is not used. Contact your local Xilinx representative for more information on using this mode.

DMA TX LocalLink Interface

This interface is compatible with the Xilinx LocalLink Specification as outlined in [Ref 6]. It is basically a synchronous, point-to-point connection that serves as a user interface to Xilinx intellectual property (IP) designs.

This unidirectional interface sends data out of the LocalLink interface for consumption by some external device, such as an EMAC. Full-duplex operation is achieved by using an RX and TX LocalLink pair simultaneously. Figure 13-2 shows the high-level connection of the TX LocalLink interface.

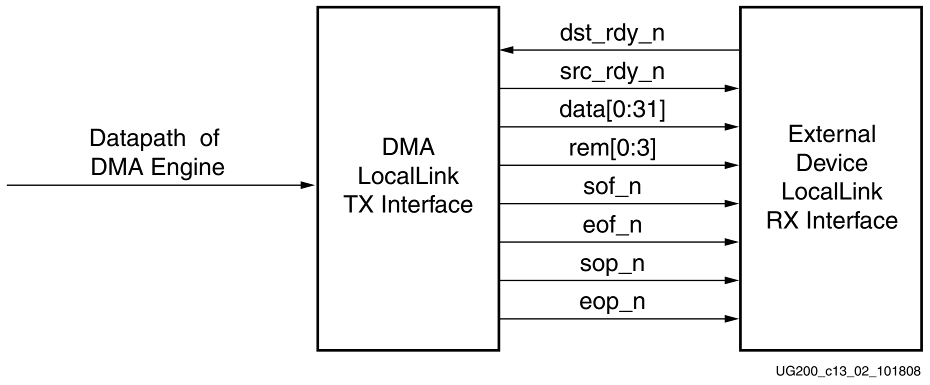


Figure 13-2: **Block Diagram of TX LocalLink Interface**

Data is sent out over the LocalLink interface as a packet, described by header, payload, and footer components. The sof_n signal initiates the header of the packet. Between the time this signal is asserted and the time the sop_n signal is asserted, the header of the packet is transmitted. Between the sop_n signal assertion and the eop_n signal assertion, the payload is transmitted. Finally, the information between eop_n and eof_n constitutes the footer. Information is deemed to be valid on the interface whenever src_rdy_n and dst_rdy_n are asserted simultaneously. The TX agent (DMA) or the RX agent can become Not-Ready at any time during transmission by deasserting src_rdy_n or dst_rdy_n.

For the DMA TX LocalLink interface, only two of the three packet components are used: the header and the payload components. During the header portion of the transmission, the control information in the first descriptor associated with a packet is transmitted over the LocalLink interface. The payload portion transfers the actual data associated with the first descriptor and possibly additional linked descriptors. When the payload completes, as indicated by the assertion of eop_n, the packet is framed immediately by the assertion of the eof_n signal. The eof_n signal is always asserted exactly one cycle after eop_n is asserted. See Figure 13-3 and Figure 13-4 for more details.

Because the packet payload is specified in number of bytes and the first complete 32-bit payload word is always sent coincident with sop_n, the last payload word might be incomplete. That is, only some of the four bytes might be valid at the end of a packet. The rem[0:3] signals are used as a mask and indicate which of the bytes in the last payload word are valid. At all other times during the transmission, rem[0:3] is driven to a value of `4'b0000`, indicating all bytes are valid. Table 13-3 shows an example of rem[0:3] for the last word of a payload (indicated by the assertion of eop_n).

Table 13-3: **TX LocalLink REM[0:3] Value During EOP_N**

Rem[0:3]	0	0	0	1
Data Bus (MSB)	[0:7]	[8:15]	[16:23]	[24:31]
	Payload Bytes			Footer Bytes

Because the footer is not relevant for TX, the single footer byte indicated in Table 13-3 is ignored by the LocalLink receiving device, because it is not part of the payload.

Figure 13-3 shows how the data packet for transmission is assembled from the descriptor information. Data is provided to the TX LocalLink module as it becomes available. Whenever data is available to send, the TX LocalLink interface can potentially assert src_rdy_n.

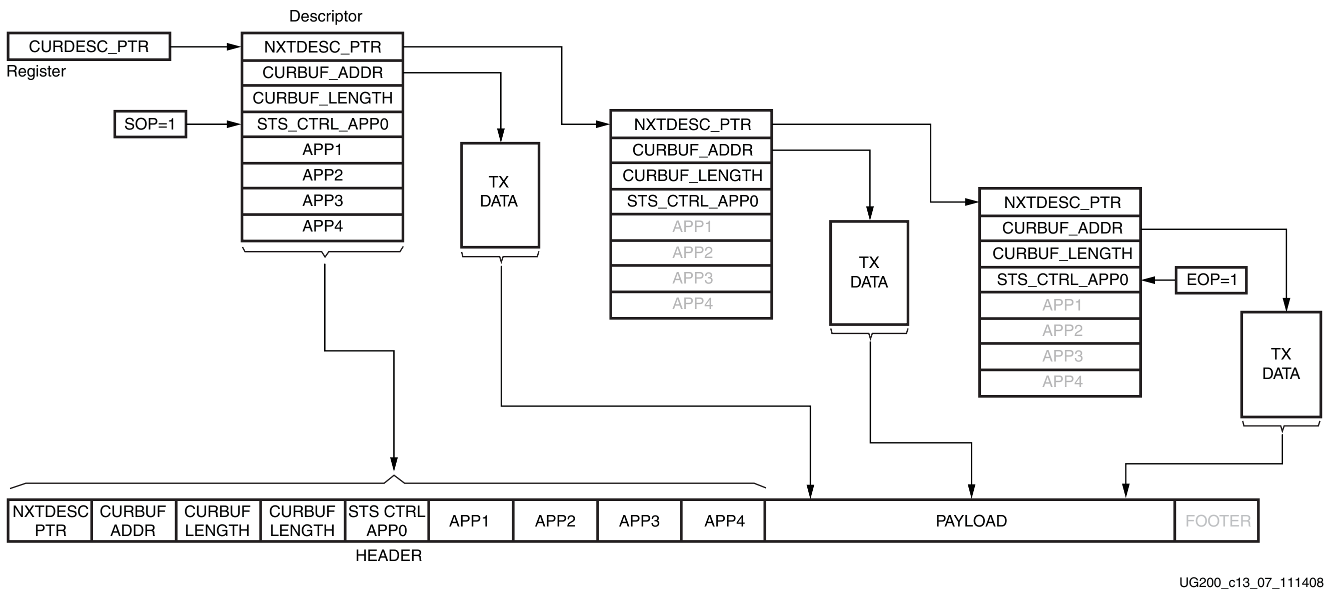


Figure 13-3: **Assembly of Transmit Data Packet**

Figure 13-4 shows a single frame from the DMA engine (source) to the FPGA logic (destination). The header length is always eight words with Word 0 first. The payload size is indicated in the header and can be variable. The End of Frame signal needs to be asserted one cycle after the End of Payload signal because the footer component is not used. When the End of Payload is asserted, the REM data indicates which bytes are valid for the last word of the payload. Each asserted REM bit (active Low) represents a valid DATA byte. REM[0] is associated with DATA[0:7], and REM[3] is associated with DATA[24:31]. Data is sampled every cycle that DST_RDY and SRC_RDY are asserted.

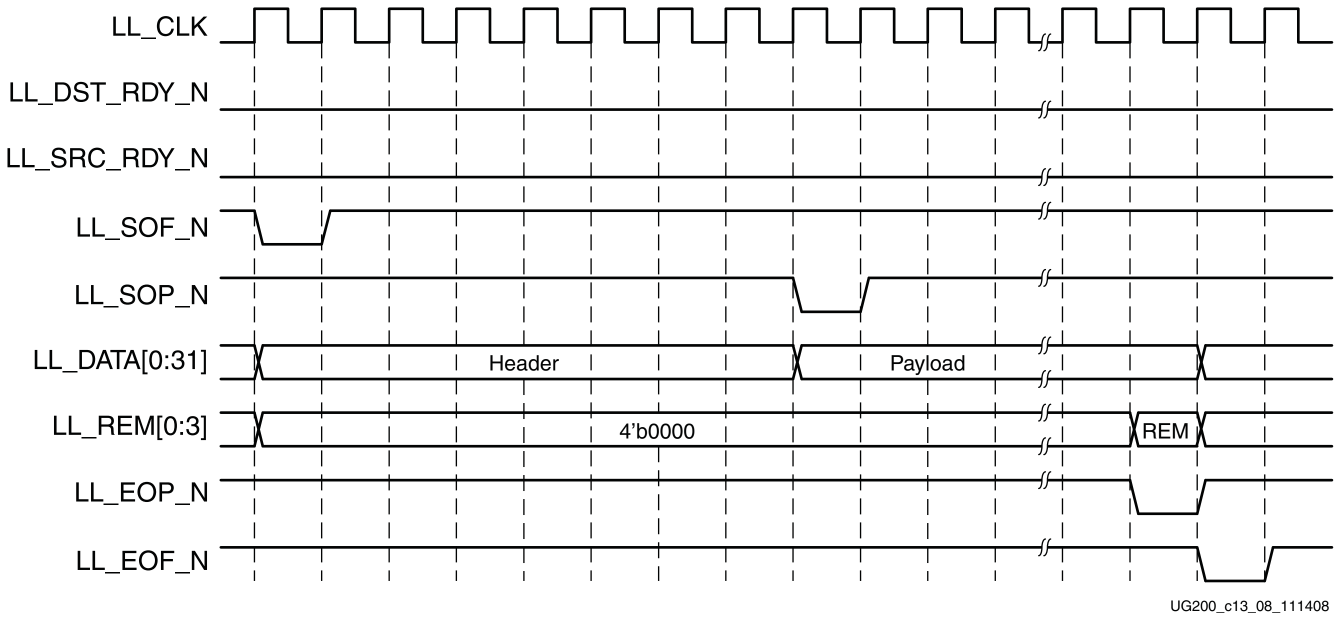


Figure 13-4: **TX Timing**

DMA RX LocalLink Interface

This interface is essentially the "other end" of the interface described in "DMA TX LocalLink Interface," page 231. The DMA RX LocalLink interface is a unidirectional interface, receiving data from the LocalLink interface sent by some external device, such as an EMAC. Full-duplex operation is achieved by using an RX and TX LocalLink pair simultaneously. Figure 13-5 shows the high-level connection of the RX LocalLink interface.

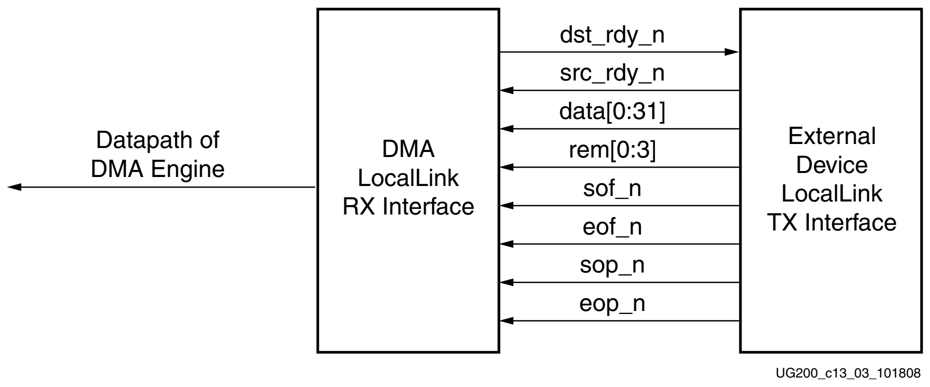


Figure 13-5: **Block Diagram of RX LocalLink Interface**

As before, data is received from the LocalLink interface as a packet, described by header, payload, and footer components. The sof_n signal initiates the header of the packet. Between the time this signal is asserted and the time the sop_n signal is asserted, the header of the packet is received. Between the sop_n signal assertion and the eop_n signal assertion, the payload is received. Finally, the information between eop_n and eof_n constitutes the footer. Information is deemed to be valid on the interface whenever src_rdy_n and dst_rdy_n are asserted simultaneously. The TX agent or the RX agent (DMA) can become "Not-Ready" at any time during reception by deasserting src_rdy_n or dst_rdy_n.

For the DMA RX LocalLink interface, only two of the three packet components are used: the payload and the footer components. Any information received during the header portion of the packet is ignored by the interface. It is therefore recommended that sop_n is asserted exactly one cycle after sof_n has been asserted. That is, do not transmit garbage header data because it is discarded.

The payload portion transfers the actual data associated with the DMA operation. When the payload completes, as indicated by the assertion of eop_n, the footer portion commences. As in the case of TX, the rem[0:3] signals indicate where the payload bytes end and where the footer bytes begin. For RX, any footer bytes indicated by the rem[0:3] signals are discarded. The following eight bytes of footer data are handled as shown in Figure 13-6. Refer to "Descriptor Format," page 229 for more details.

After these eight words are received, the packet should be framed immediately by the assertion of the eof_n signal. This signal is required to always be asserted coincident with the last of the eight footer words.

Figure 13-6 shows how the received data packet is copied to memory, based on the descriptors, and how the descriptors are updated.

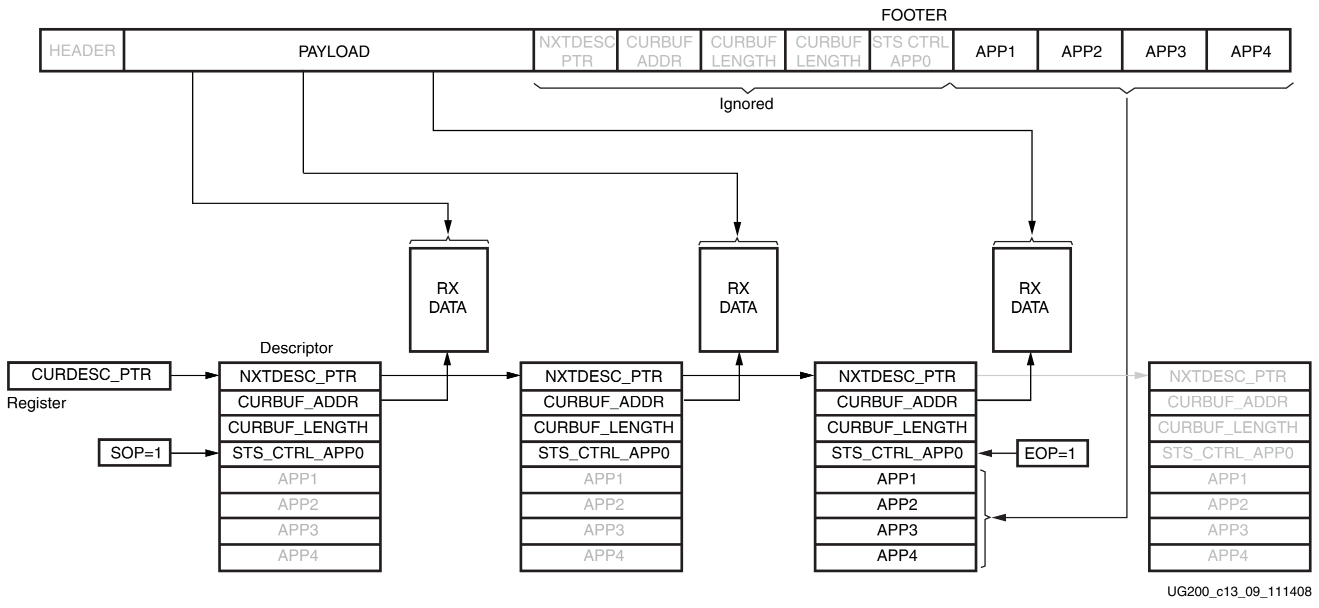


Figure 13-6: **LocalLink Frame and the Descriptor Chain for an RX Operation**

Whenever sop_n or eop_n is asserted by the peripheral that is transmitting data over this LocalLink interface, the corresponding status bit is set in the corresponding RX_STATUS_REG DCR, and is later updated to the current descriptor memory. Figure 13-7 shows a single frame from the FPGA logic (source/transmitter) to the DMA engine (destination/receiver).

The footer length is always 8 words with Word 0 first. The payload size is indicated in the footer and can be variable. The Start of Payload signal needs to be asserted one cycle after the Start of Frame signal because the header component is not used. Just as for TX, when the End of Payload is asserted, the REM data indicates which bytes are valid for the last word of the payload.

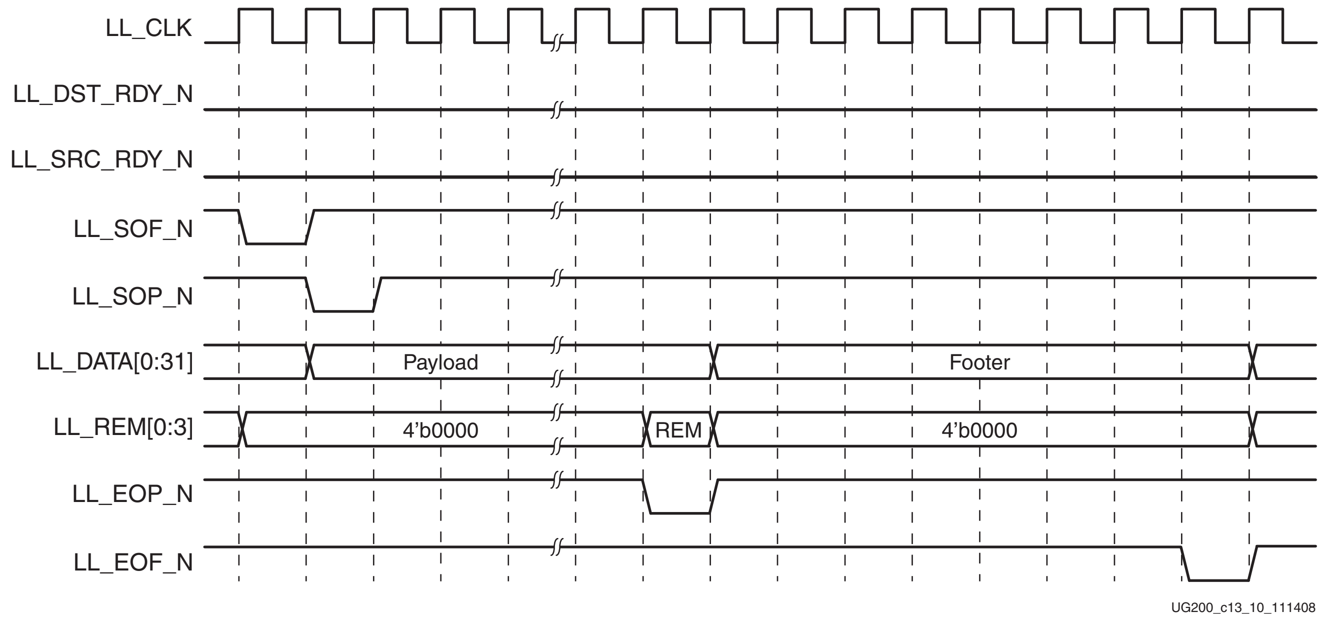


Figure 13-7: **Single Frame to DMA Engine**

Masking of Application Data Update

The MaskEnable mode can be enabled by setting a DCR bit called appMasken. When this mode is active, the first three words of the footer's second quadword (words 4, 5, and 6) can be selectively updated to memory by using a special encoding in the last corresponding rem[0:3] signal values. The rem bits are active Low. A Low on a rem bit means update the word. The following restrictions apply when using this mode:

1. The last word of the footer, Word 7, must always be updated to memory.
2. The written words must be contiguous. For example, Word 5 and Word 7 cannot be updated without updating Word 6, too. Any value of rem[0:3] other than `4'b0000` is illegal when this mode is enabled.

Table 13-4 provides further clarification.

Table 13-4: **RX LocalLink REM[0:3] during Masking Application Data Mode**

Footer Word #	REM[0:3]	Resultant Action/Comments
Word 0	`0000`	Must be `0000`. The mask is not valid for this word. The word is always updated to memory.
Word 1	`0000`	Must be `0000`. The mask is not valid for this word. The word is always updated to memory.
Word 2	`0000`	Must be `0000`. The mask is not valid for this word. The word is always updated to memory.

Table 13-4: **RX LocalLink REM[0:3] during Masking Application Data Mode *(Cont'd)***

Footer Word #	REM[0:3]	Resultant Action/Comments
Word 3	`0000`	Must be `0000`. The mask is not valid for this word. The word is always updated to memory.
Word 4	{M3,M2,M1,M0}	If ({M3,M2,M1,M0} = `4'b1111`)), do not update this word to memory If ({M3,M2,M1,M0} = `4'b0000`)), update this word to memory
Word 5	{M3,M2,M1,M0}	If ({M3,M2,M1,M0} = `4'b1111`)), do not update this word to memory If ({M3,M2,M1,M0} = `4'b0000`)), update this word to memory
Word 6	{M3,M2,M1,M0}	If ({M3,M2,M1,M0} = `4'b1111`)), do not update this word to memory If ({M3,M2,M1,M0} = `4'b0000`)), update this word to memory
Word 7	`0000`	Must be `0000`. The mask is not valid for this word. The word is always updated to memory.

DMA Addressing Limitation

The descriptor tables are configured as blocks of eight words, with the buffer address and next descriptor pointers described by 32-bit fields. However, the processor block supports 36-bit addresses, and the DMA engine extends the 32-bit address to 36 bits by statically setting the values of the four most significant bits of the address to 0.

Interrupt Mechanism

There are two interrupt pins (INT) per DMA engine, one for RX and one for TX. Eight internal events can cause the INT pins to be asserted. These events are:

1. TX Delay Timer times out
2. RX Delay Timer times out
3. TX Interrupt Coalescing Counter reaches zero
4. RX Interrupt Coalescing Counter reaches zero
5. Maskable TX Error Condition
6. Maskable RX Error Condition
7. Non-Maskable TX Error Condition
8. Non-Maskable RX Error Condition

There are separate interrupt enable bits for the RX channel (RxIrqEn) and the TX channel (TxIrqEn). Additionally, the RX and TX delay timers, the RX and TX coalescing counters, and the RX and TX Error Irqs can be independently enabled or disabled by DCR control fields. This way the interrupt sources of interest can be controlled precisely. The two non-maskable interrupts do not have individual interrupt enable controls. Setting these bits causes a non-recoverable serious system issue. The status of all the interrupt sources can always be read in the interrupt register. The enable bits control whether or not that particular interrupt source causes the interrupt pin to be asserted to the CPU or not (excluding the non-maskable interrupts).

Figure 13-8 shows the higher level connection of the interrupt events to the interrupt output pin.

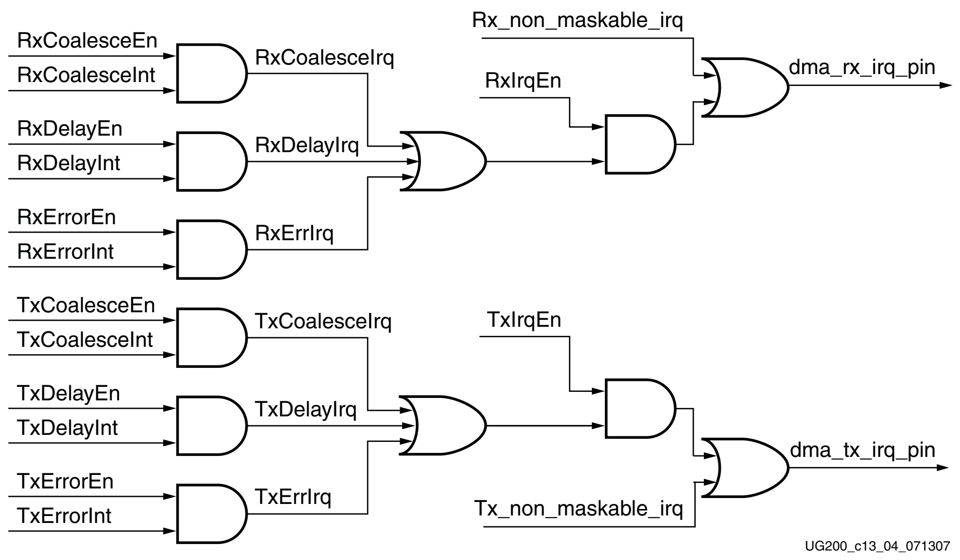


Figure 13-8: **DMA Engine Interrupt Scheme**

When the CPU is interrupted, it reads the corresponding TX or RX DCR Interrupt register to find out the source of the interrupt. The interrupt register has a separate interrupt bit for each of the three interrupt sources, as shown in Figure 13-8. The interrupt can then be cleared by writing a "1" to the corresponding field(s) in the DCR Interrupt register. This does not apply to the non-maskable interrupts.

Maskable Error Interrupts

The Maskable Error interrupt bit is set if any of the following conditions are true:

1. The Delay Timer Interrupt Counter overflows (this event can be disabled).
2. The Coalescing Interrupt Counter overflows (this event can be disabled).
3. The Current Descriptor Pointer is written while the channel is busy.

No further action is taken by the DMA other than the setting of the Error bit.

Non-Maskable Error Interrupts

The Non-Maskable Error interrupt bit is set if any of the following conditions are true:

1. The Internal PLB subsystem issues a rdDataErr to the DMA.
2. The Internal PLB subsystem issues a wrDataErr to the DMA.

In both cases, the LocalLink interface is frozen immediately to prevent any data corruption from occurring. The only way to recover from this situation is to reset the DMA engine (either power down or software/hardware DMA reset).

Delay Timer

Figure 13-9 shows the mechanism used for the TX Delay Timer interrupt generation.

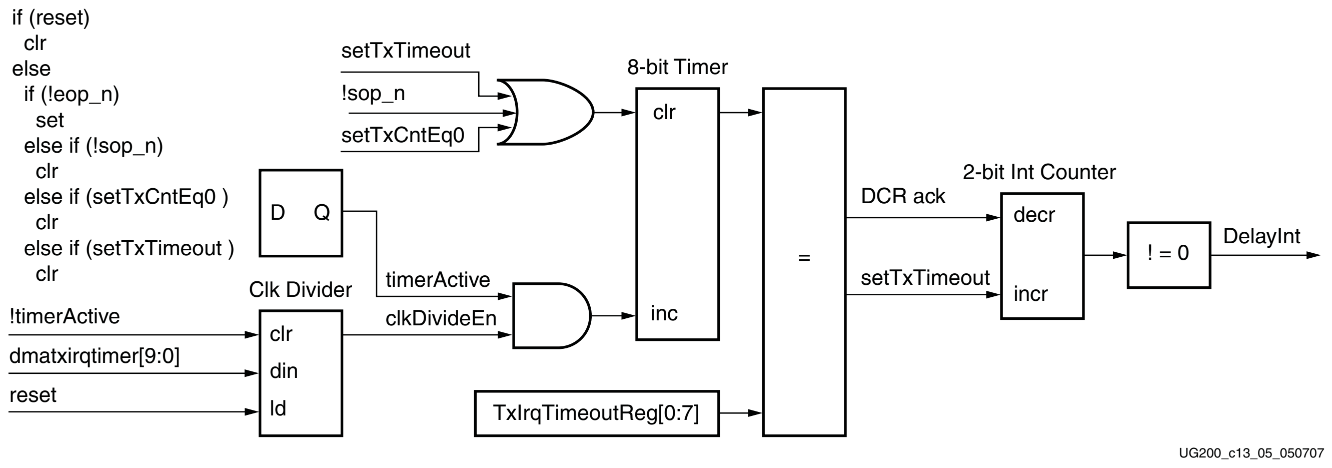


Figure 13-9: **Delay Timer Interrupt Scheme**

The delay timer is needed because interrupt coalescing is used in the DMA. For example, if the RX coalescing counter is set to 10, every 10 packets received will generate an interrupt. But assume five packets are received on the Ethernet and then the channel goes idle (no traffic). The CPU never processes the five packets because no interrupt is generated, and this interrupt happens only when (or if) five more packets arrive. To avoid this latency, a timer is needed that will fire when all of the following are true:

- a packet has been received
- some (software settable) time has elapsed
- no more packets are received during this time

The only purpose of this timer is to avoid large latencies in the received packet (which is sitting in main memory by this time) from being processed by the CPU when there is non-continuous traffic on the wire.

The Clock Divider module uses a 10-bit tie-off value (embedded processor block attribute DMAn_TXIRQTIMER or DMAn_RXIRQTIMER) to determine how many LocalLink clock cycles to count before generating a single clkDivideEn pulse. A typical LocalLink clock speed of 200 MHz translates to a 5.12 µs clkDivideEn period. Therefore the 8-bit timer can count up to a maximum of 256 * 5.12 = 1.3 ms, before generating an interrupt.

When the coalescing counter fires, the delay timer is automatically cleared.

Interrupt Coalescing Counter

The interrupt coalescing counter is an additional mechanism used for interrupt handing. It relieves the CPU from having to service an interrupt at the end of every packet. Instead, a preloadable number of interrupt events (up to 256) generates a single interrupt to the CPU. Figure 13-10 shows the mechanism used for the TX coalescing counter interrupt generation.

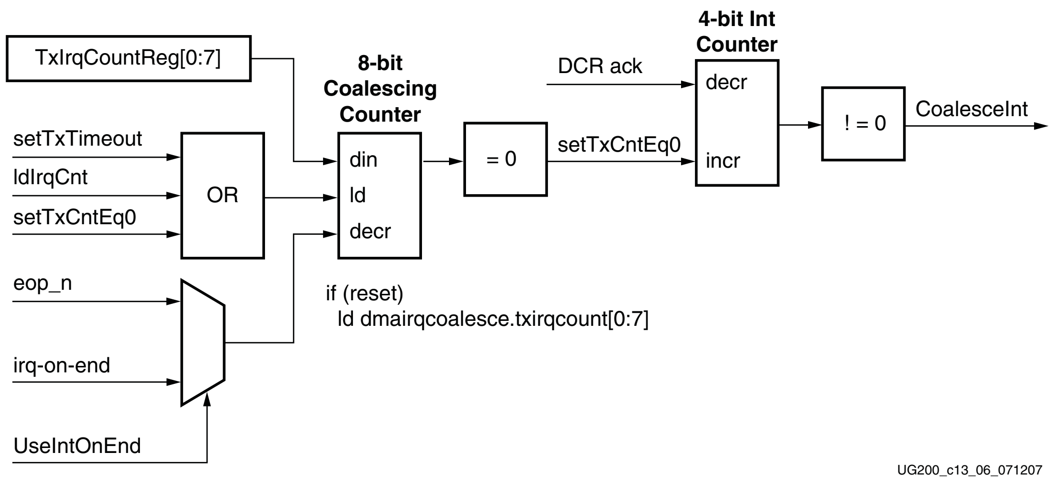


Figure 13-10: **Coalescing Counter Interrupt Scheme**

On reset, the dmairqcoalesce.txirqcount[0:7] value is used to load the coalescing counter. The CoalesceCounterValue field in the DCR TX Interrupt Register can be subsequently programmed with any eight-bit value. On every eop or irq-on-end (selected by the UseIntOnEnd field in the DCR), the counter decrements. When the coalescing counter reaches 0, the DMA increments the four-bit int counter. Whenever the four-bit int counter is non-zero, it generates an interrupt to the CPU (if the irq enable bit of the respective channel is set). Whenever the interrupt is acknowledged (DCR write of 1), the four-bit int counter is decremented. The contents of the CoalesceCounterValue field are reloaded when the four-bit int counter is incremented.

The CPU also can force the counter to load the contents of the CoalesceCounterValue field by writing to the ldIrqCnt field of the TX Channel Control register.

This approach is used because there are not enough unique DCR addresses to allow a DCR write to a unique address for loading the TX and RX counters.

When the delay timer fires, the coalescing counter is automatically reloaded.

Dynamic Descriptor Appending

The DMA controller allows the CPU to allocate more work to the DMA channel dynamically. Two DCRs are defined for this purpose per DMA channel: the Current Descriptor Pointer and the Tail Descriptor Pointer. A typical software sequence to start a DMA operation is:

1. Set up the descriptors in memory.
2. Write the Current Descriptor Pointer register with the first descriptor base address in memory.
3. Write the Tail Descriptor Pointer register with the last descriptor base address.

Writing to the Tail Descriptor Pointer register triggers a start for the DMA channel. From this time on, software should NOT write the Current Descriptor Pointer register unless it is certain that the descriptor chain has stopped executing. The descriptor fetching and executing continue until the StopOnEnd bit is encountered or until the Current Descriptor Pointer register equals the Tail Descriptor Pointer register. At this point, the Descriptor state machine of the channels returns to the IDLE state. The DMA checks if the Current

Descriptor Pointer register equals the Tail Descriptor Pointer register *after* it has executed the current descriptor. Software can restart operation by repeating the three steps above.

For software to add more descriptors *dynamically* to the existing chain, it simply writes a new address to the Tail Descriptor Pointer register at any time during the DMA process. It does NOT have to wait for the DMA channel to be IDLE to do this. Software can write the Tail Descriptor Pointer even if the DMA is IDLE. In this case, the DMA continues to execute from where it left off. See "Software/Device Driver Considerations" for more information.

DMA Engine Reset

The DMA controller provides the capability to reset a particular DMA engine (both RX and TX channels simultaneously) whenever a lockup situation arises, typically over the LocalLink interface. There are two separate mechanisms available for this purpose: a hardware reset and a software reset.

Hardware Engine Reset Mechanism

With this interface, all inputs and outputs are synchronous to the LocalLink clock. This scheme is useful if hardware needs to reset the engine directly. Two pins are added per DMA engine: a rst_engine_req input and a rst_engine_ack output. When an engine reset is required, the rst_engine_req pin is asserted for one cycle. The DMA engine immediately asserts the rst_engine_ack output pin. This output can be used as an external logic reset. It then proceeds to shut down operations and waits for all the pipelined commands to flush through the processor block before deasserting the rst_engine_ack output pin. This indicates that the engine reset operation is completed.

Software Engine Reset Mechanism

The DMA Control register contains a software reset bit. When a 1 is written to the SwReset bit, it initiates the reset sequence for that particular engine. At the same time, the DMALLRSTENGINEACK output pin is asserted, synchronous to the LocalLink clock. This output can be used as an external logic reset. Software needs to now poll the SwReset bit until it is sampled cleared, which indicates that the reset sequence has completed and the pipeline is flushed. Simultaneously with the SwReset bit being cleared, the DMALLRSTENGINEACK pin is automatically deasserted.

Whenever the DMA engine reset function is used, there is no guarantee that the current descriptor completed correctly. The assumption should be that the descriptor did not complete and it should be restarted again using the normal CPU technique for kicking off a new DMA operation.

Software/Device Driver Considerations

Software applications use the DMA controller by first setting up a linked list of descriptors in memory, and then writing the addresses of the head and tail of this list to TX_CURDESC_PTR and TX_TAILDESC_PTR, respectively (or to RX_CURDESC_PTR and RX_TAILDESC_PTR). This list of descriptors can correspond to one or more packets on the LocalLink interface. As each descriptor is processed by the DMA controller, status bits are updated in memory by the DMA controller, and the software application running on the processor can read these bits to monitor the progress. When the DMA controller completes processing of a descriptor, it sets the DMA_COMPLETED status bit in the descriptor to 1. The software application then processes the data received for an RX transaction. Finally, the software application frees up the memory used by the descriptor. So a descriptor has the following life cycle:

1. It is first created/allocated in memory.
2. Then it is pre-processed by the software application to set up data and control values, and attached to the descriptor chain.
3. Then it is handed over to the DMA controller hardware.
4. Finally, it is post-processed and freed.

Software applications typically send and receive packets repeatedly over a period of time, and the Dynamic Descriptor Appending mode (or Tail_pointer mode) allows the above steps to be done much more efficiently by organizing the descriptors into a ring instead of a simple chain, as shown in Figure 13-11.

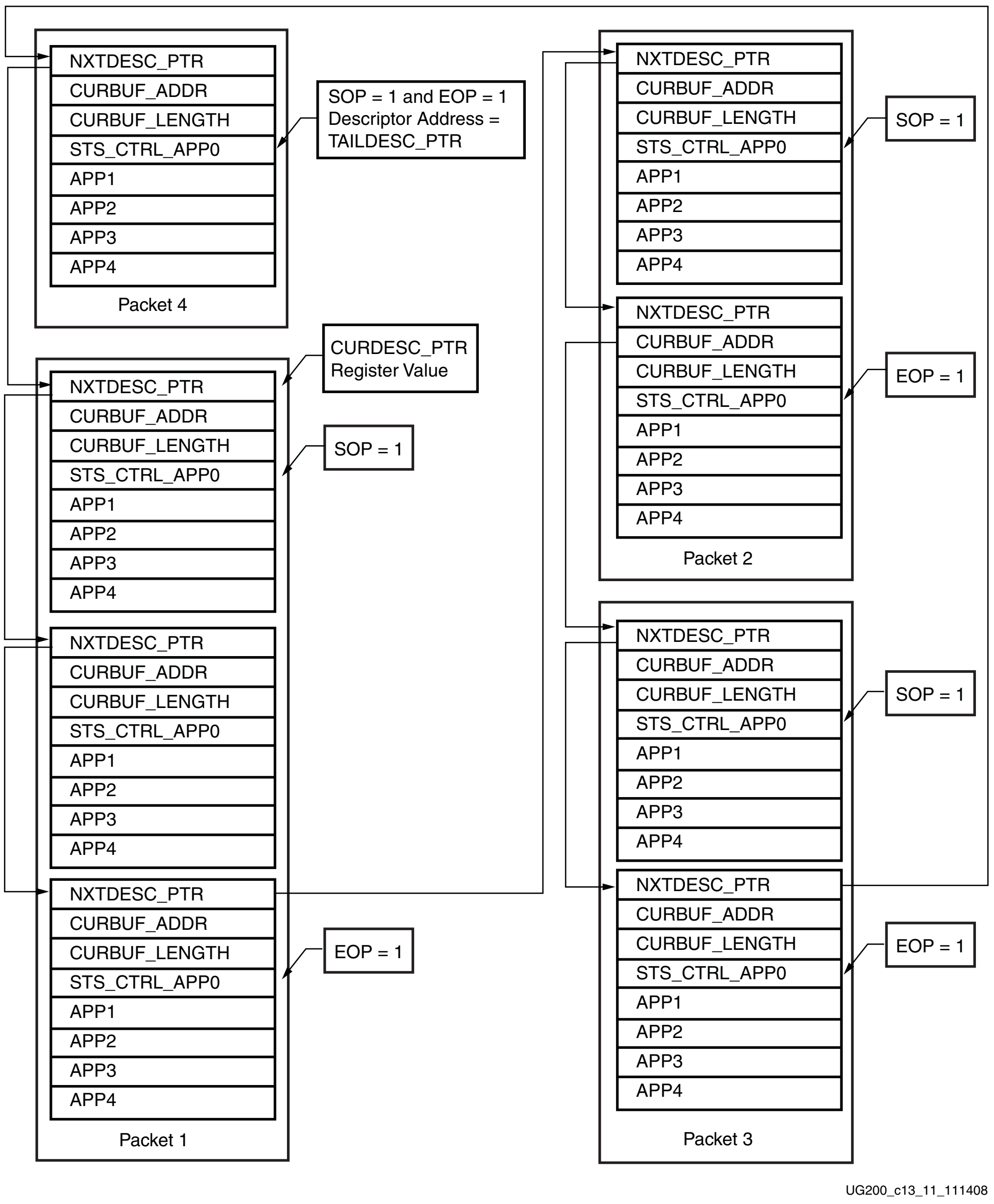


Figure 13-11: **Descriptor Organization (Ring)**

The ring typically has more descriptors than needed for a single transaction on an RX or TX channel. For the very first transaction, the software application writes both the CURDESC_PTR and TAILDESC_PTR to the DMA controller. For subsequent transactions, the software application simply writes the TAILDESC_PTR after setting up the next set of descriptors. Figure 13-12 shows a snapshot of such a ring at some point in its operation.

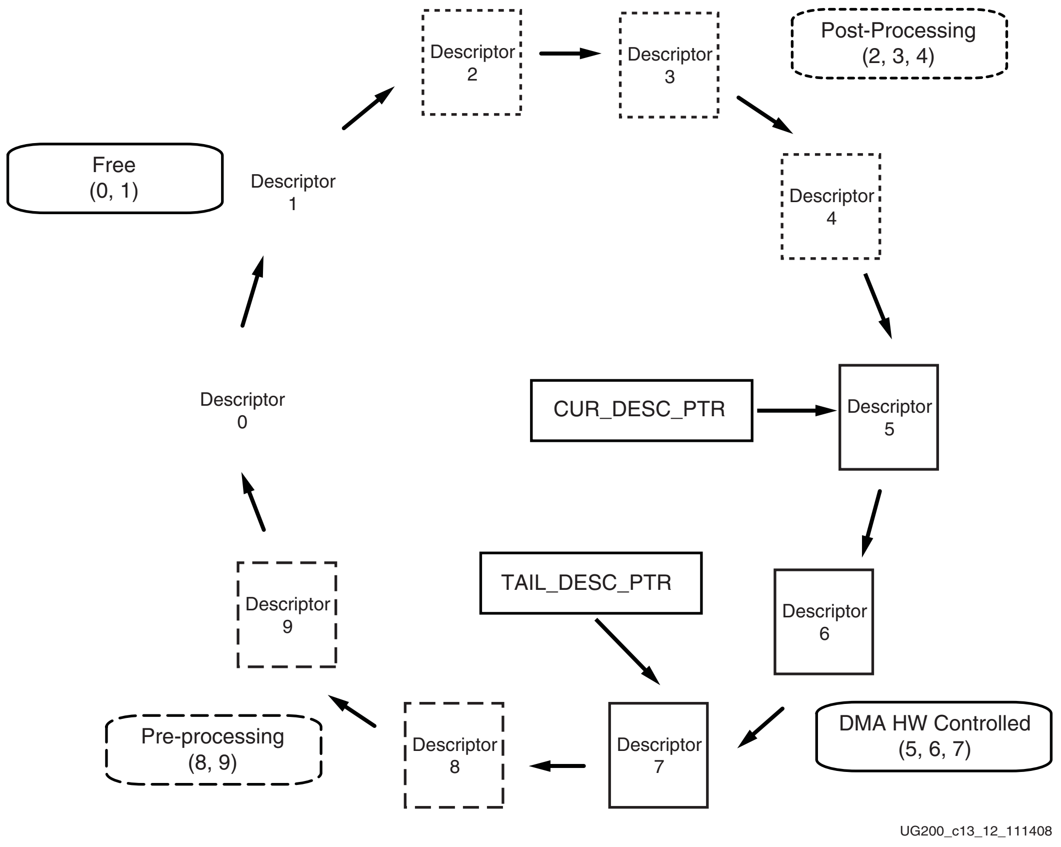


Figure 13-12: **Descriptor Ring Snapshot**

Figure 13-12 shows a descriptor ring with some descriptors in each possible state. The descriptors that are under the control of the DMA controller hardware are framed by CURDESC_PTR and TAILDESC_PTR. CURDESC_PTR is set initially by the software application, and thereafter updated by the DMA controller hardware to point to the current descriptor that is being processed by the DMA controller hardware.

Descriptors 8 and 9 are in the pre-processing stage, indicating that the software application has filled in the control and data values for these descriptors for the next transaction but not handed them over to the DMA controller hardware yet. When the software application is done with updating the descriptors for the next transaction, it simply updates the TAILDESC_PTR to point to the last descriptor updated.

Descriptors 2, 3, and 4 are in the post-processing state, indicating that the DMA controller hardware has finished processing them and has set their DMA_COMPLETED status bit to 1. The software application marks them as free, after doing whatever post-processing is needed. This scheme improves the efficiency of the DMA transfer process in several ways:

- By allocating a separate descriptor ring for each DMA channel, the software application ensures that descriptors are allocated and freed in the same order.

- Instead of making the operating system software handle the allocation and freeing of memory for each transaction, the software application/driver uses a lightweight allocation/free scheme that simply updates a few fields in pre-allocated descriptors.
- Instead of writing two hardware registers (CURDESC_PTR and TAILDESC_PTR) for each transaction, the software application writes just one register (TAILDESC_PTR) for all but the first transaction.

Specific additional considerations are:

- The CURDESC_PTR register as well as the NXTDESC_PTR field of the descriptor *must* be eight-word aligned. The TAILDESC_PTR can have any byte alignment.
- To reliably allow appending descriptors, software must not modify the STATUS/CONTROL field of any descriptor that it makes visible to the hardware.
- It is very difficult for software to manage descriptors in cacheable memory. As a result of writing any byte in a descriptor in cacheable memory, the CPU might then write the whole 32-byte descriptor, which could interfere with updates the hardware has done.
- The hardware can fetch the last descriptor that software makes available long before using it. Thus the NXTDESC_PTR pointer in the descriptor to which the TAILDESC_PTR points must point to what will be the next updated descriptor by software because that pointer cannot be changed. The descriptor ring scheme described earlier avoids this problem.
- For TX, it is not recommended to set a StopOnEnd bit in a descriptor without the EOP bit being set (or without the CURDESC_PTR == TAILDESC_PTR comparison being TRUE).
- Each DMA engine must operate in a 4 Gbyte segment of memory. It is software's responsibility to ensure that a DMA operation does not roll over into a new 4 Gbyte segment. Refer to "DMA Addressing Limitation," page 236 for details.
- If the CPU is interrupted due to a descriptor being completed (or if the CPU reads the DMA engine status register), the CPU can process completed buffer descriptors by following the descriptor chain from the last completed buffer descriptor (BD). Refer to "Implementation Note" for this use mode.
- The descriptor buffer length field should never be programmed to a value of 0.
- There are two methods to stop the descriptor fetching process:
  - If the descriptor status field, StopOnEnd, is set.
  - If the CURDESC_PTR register equals the TAILDESC_PTR register and the current descriptor has been completed.
  - If either of these conditions is TRUE, the DMA channel stops the descriptor fetching process. In this case, a subsequent write to the Tail Descriptor Pointer register restarts the fetching process.

Implementation Note

For RX DMA, a descriptor update occurs after the payload is successfully received from the LocalLink and written to memory. This update always consists of two separate single quadword writes to memory. The first quadword write is essentially the descriptor "status" byte, plus three bytes of user application data. The second quadword write is four words of user application data. One of the fields in the status byte, the "completed" field, indicates that a descriptor is completed by the DMA.

When software reads the updated "status" from memory, it is not sure if the second quadword has been written to memory yet. Technically, the packet is completed when the second quadword is updated to memory, because it might contain information used by software (for example, packet size).

In order to overcome this behavior, a word of the second descriptor quadword in memory must now be pre-initialized to a unique pattern, for example, `32'hFFFF_FFFF`. When the second quadword is updated by the DMA, this pre-initialized word is now overwritten. The application must ensure that the word is overwritten with a different pattern. The driver now uses this fact to indicate if a packet has been completed or not.

Programming Interface and Registers

The register address map and register details presented in this section are identical for all DMA engines.

DCR Address Map

Table 13-5 lists the address map for the DCRs.

Table 13-5: **DCR Address Map**

DCR Addresses	Mnemonic	Register Description	Direction
`0x80,0x98,0xB0,0xC8`	TX_NXTDESC_PTR	TX Next Descriptor Pointer	RW(1)
`0x81,0x99,0xB1,0xC9`	TX_CURBUF_ADDR	TX Current Buffer Address Register	RW(1)
`0x82,0x9A,0xB2,0xCA`	TX_CURBUF_LENGTH	TX Current Buffer Length Register	RW(1)
`0x83,0x9B,0xB3,0xCB`	TX_CURDESC_PTR	TX Current Descriptor Pointer	RW
`0x84,0x9C,0xB4,0xCC`	TX_TAILDESC_PTR	TX Tail Descriptor Pointer	RW
`0x85,0x9D,0xB5,0xCD`	TX_CHANNEL_CTRL	TX Channel Control Register	RW
`0x86,0x9E,0xB6,0xCE`	TX_IRQ_REG	TX Interrupt Register	RD-ACK
`0x87,0x9F,0xB7,0xCF`	TX_STATUS_REG	TX Status Register	RW(1)
`0x88,0xA0,0xB8,0xD0`	RX_NXTDESC_PTR	RX Next Descriptor Pointer	RW(1)
`0x89,0xA1,0xB9,0xD1`	RX_CURBUF_ADDR	RX Current Buffer Address Register	RW(1)
`0x8A,0xA2,0xBA,0xD2`	RX_CURBUF_LENGTH	RX Current Buffer Length Register	RW(1)
`0x8B,0xA3,0xBB,0xD3`	RX_CURDESC_PTR	RX Current Descriptor Pointer	RW
`0x8C,0xA4,0xBC,0xD4`	RX_TAILDESC_PTR	RX Tail Descriptor Pointer	RW
`0x8D,0xA5,0xBD,0xD5`	RX_CHANNEL_CTRL	RX Channel Control Register	RW
`0x8E,0xA6,0xBE,0xD6`	RX_IRQ_REG	RX Interrupt Register	RD-ACK
`0x8F,0xA7,0xBF,0xD7`	RX_STATUS_REG	RX Status Register	RW(1)
`0x90,0xA8,0xC0,0xD8`	DMA_CONTROL_REG	DMA Control Register	RW

Notes:

1. These registers are loaded from the descriptors and updated dynamically by the DMA engine. As such, they should not be written during normal operation. Writing them is made available for debug purposes only.
2. See also the DMA enable and DMA priority fields of the SPLB 0 and SPLB 1 configuration registers (CFG_PLBS0/1) Table 4-6, page 108 in Chapter 4.

DCR Descriptions

This section describes the fields within the DCRs in detail.

TX Next Descriptor Pointer

Figure 13-13 shows the TX Next Descriptor Pointer. Table 13-6 defines the bits in this pointer.

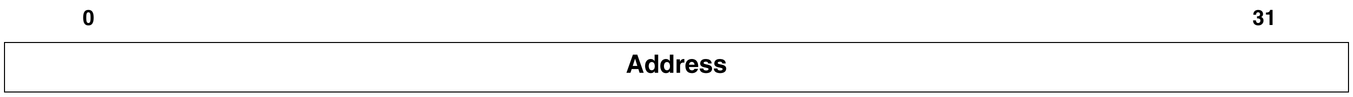


Figure 13-13: **TX Next Descriptor Pointer**

Table 13-6: **Bit Description for TX Next Descriptor Pointer**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	Contains the address of the next descriptor to be fetched. Must be eight-word aligned.

TX Current Buffer Address Register

Figure 13-14 shows the TX Current Buffer Address register. Table 13-7 defines the bits in this register.

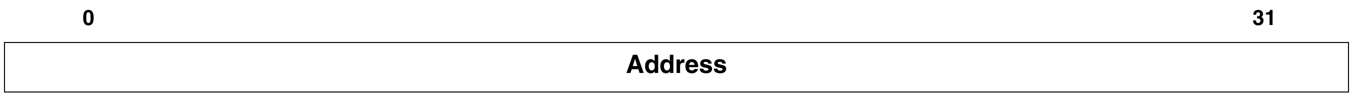


Figure 13-14: **TX Current Buffer Address**

Table 13-7: **Bit Description for TX Current Buffer Address**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	Contains the current payload address. This field changes dynamically when the DMA is operating. This address is a byte address.

TX Current Buffer Length Register

Figure 13-15 shows the TX Current Buffer Length register. Table 13-8 defines the bits in this register.

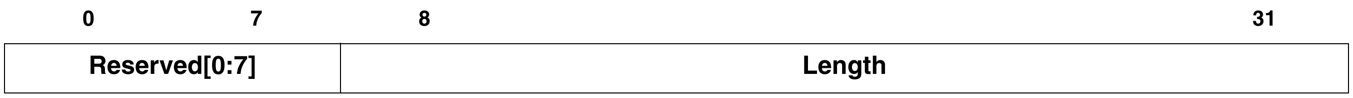


Figure 13-15: **TX Current Buffer Length Register**

Table 13-8: **Bit Description for the TX Current Buffer Length Register**

Bit	Name	Default	Description
[8:31]	Length	`24'h00_0000`	Contains the remaining 24-bit payload length to be transferred. This field changes dynamically when the DMA is operating.
[0:7]	Reserved		

TX Current Descriptor Pointer

Figure 13-16 shows the TX Current Descriptor Pointer register. Table 13-9 defines the bits in this register.

0 31
Address

Figure 13-16: **TX Current Descriptor Pointer**

Table 13-9: **Bit Descriptions for the TX Current Descriptor Pointer**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	Contains the address of the currently executing descriptor. Must be eight-word aligned.

TX Tail Descriptor Pointer

Figure 13-17 shows the TX Tail Descriptor Pointer register. Table 13-10 defines the bits in this register.

0 31
Address

Figure 13-17: **TX Tail Descriptor Pointer**

Table 13-10: **Bit Descriptions for the TX Tail Descriptor Pointer**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	These bits contain the address of the last descriptor to be fetched. When this register is written to, it initiates a fetch from the address pointed to by the TX Current Descriptor Pointer register. This register can be updated dynamically, while the DMA channel is busy. The control register field, TailPtrEn, must be set for this feature to be enabled.

TX Channel Control Register

Figure 13-18 shows the TX Channel Control register. Table 13-11 defines the bits in this register. This register controls operation for the TX channel only. The default value of this register is user configurable through the DMAn_TXCHANNELCTRL parameters on the processor block instantiation in the user's design.

24	25 28	29	30	31
IrqEn	Reserved[25:28]	IrqErrorEn	IrqDelayEn	IrqCoalesceEn

16 19	20	21	22	23
MsbAddr[0:3]	Reserved	Use1BitCnt	UseIntOnEnd	LdIrqCnt

8 15
IrqCount[0:7]

0 7
IrqTimeout[0:7]

Figure 13-18: **TX Channel Control Register**

Table 13-11: **Bit Descriptions for the TX Channel Control Register**

Bit	Name	Description
[31]	Coalescing Mechanism Interrupt Enable	Enable (1) or disable (0) the Coalescing interrupt mechanism.
[30]	Delay Timer Mechanism Interrupt Enable	Enable (1) or disable (0) the Delay Timer interrupt mechanism.
[29]	Error Detect Mechanism Interrupt Enable	Enable (1) or disable (0) the Error Detection interrupt mechanism.
[25:28]	Reserved	Reserved
[24]	Master Interrupt Enable	When set, this bit indicates that the DMA TX channel is enabled to generate interrupts to the CPU. This is the *master* enable for the TX channel. Individual interrupt sources can be enabled or disabled separately.
[23]	Load the Interrupt Coalescing Counter	Writing a 1 to this field forces the loading of the Interrupt Coalescing counters from the DCR IrqCount[0:7] field. This bit is self-clearing.
[22]	Use the Interrupt-On-End Mechanism	• 1: Select the interrupt-on-end mechanism for interrupt coalescing. • 0: Select the eop mechanism for interrupt coalescing.
[21]	Use 1-bit Interrupt Counters	When this bit is enabled, the four-bit Interrupt Coalescing counter and two-bit Delay Timer counters are forced to be one-bit only. For certain device driver applications, this is a desirable use model.
[20]	Reserved	Reserved
[16:19]	Msb Address	These bits contain the statically assigned, most-significant four bits of the DMA address. This field must be all zeros.
[8:15]	Interrupt Coalescing Count Value	These bits contain the eight-bit value to be preloaded into the TX interrupt coalescing counter. They are loaded into the counter when a write to the TX LdIrqCnt field is performed and subsequently reloaded whenever the Count reaches 0.
[0:7]	Interrupt Delay Time-out Value	These bits hold the compare value for the TX interrupt delay timer. The value in this field is compared to the TX Irq Delay Timer output. When they are equal, a TX interrupt event is generated.

TX Interrupt Register

This register contains the interrupt status bits for the TX channel as well as read-only status for the TX coalescing and delay timer counters and timers. There are three regular interrupt sources: ErrorIrq, DelayIrq, and CoalesceIrq. There are two non-maskable interrupts (NMI): PlbRdErr and PlbWrErr.

A regular interrupt can be acknowledged (and hence cleared if the corresponding counter equals 0), by writing a "1" to the respective interrupt status bit in this register. The NMIs can only be cleared by issuing a reset to the DMA (hard or soft).

Figure 13-19 shows the TX Interrupt register. Table 13-12 defines the bits in this register.

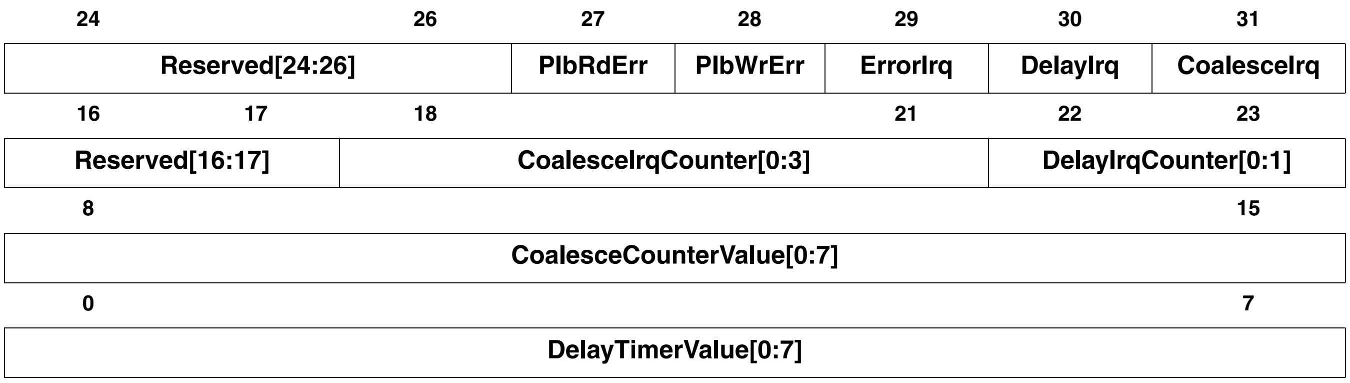


Figure 13-19: **TX Interrupt Register**

Table 13-12: **Bit Descriptions for the TX Interrupt Register**

Bit	Name	Default	Description
[31]	Coalescing Counter Interrupt	`1'b0`	When this bit is 1 the TX DMA channel has a pending interrupt because of a TX Coalescing interrupt counter greater-than-0 condition. This bit is ORed with the two other TX interrupt bits and ANDed with the TX Interrupt Enable bit to produce the TX Irq pin. Even if the TxIrqEn bit is disabled, software can still poll this bit. Acknowledging a TX interrupt due to a coalescing counter condition is accomplished by writing a 1 to this bit. This action decrements the TX Coalescing interrupt counter.
[30]	Delay Timer Interrupt	`1'b0`	When this bit is 1, the TX DMA channel has a pending interrupt because of a TX Delay Timer interrupt counter greater-than-0 condition. This bit is ORed with the two other TX interrupt bits and ANDed with the TX Interrupt Enable bit to produce the TX Irq pin. Even if the TxIrqEn bit is disabled, software can still poll this bit. Acknowledging a TX interrupt due to a Delay Timer counter condition is accomplished by writing a 1 to this bit. This action decrements the TX Delay Timer interrupt counter.
[29]	Error Interrupt	`1'b0`	When this bit is 1, the TX DMA channel has a pending interrupt because of a TX error that has occurred. This bit is ORed with the two other TX interrupt bits and ANDed with the TX Interrupt Enable bit to produce the TX Irq pin. Even if the TxIrqEn bit is disabled, software can still poll this bit. Acknowledging a TX interrupt due to an error is accomplished by writing a 1 to this bit. This action clears this bit.
[28]	PLB Write Error Non-Maskable Interrupt	`1'b0`	When this bit is 1, the TX DMA channel has received an error from the PLB due to a PLB write operation. This serious error causes the DMA to freeze the LocalLink interface as soon as it receives this indication from the crossbar. This bit can only be cleared by resetting the DMA (hard or soft).
[27]	PLB Read Error Non-Maskable Interrupt	`1'b0`	When this bit is 1, the TX DMA channel has received an error from the PLB due to a PLB read operation. This serious error causes the DMA to freeze the LocalLink interface as soon as it receives this indication from the crossbar. This bit can only be cleared by resetting the DMA (hard or soft).
[24:26]	Reserved		

Table 13-12: **Bit Descriptions for the TX Interrupt Register *(Cont'd)***

Bit	Name	Default	Description
[22:23]	Delay Timer Interrupt Counter		This read-only field is useful for debug purposes. It contains the value of the two-bit counter used to store the number of TX Delay Timer interrupts that are outstanding.
[18:21]	Coalescing Interrupt Counter		This read-only field is useful for debug purposes. It contains the value of the four-bit counter used to store the number of TX coalescing counter interrupts that are outstanding.
[16:17]	Reserved		
[8:15]	Coalescing Counter Value		This read-only field is useful for debug purposes. It contains the value of the eight-bit Coalescing Counter.
[0:7]	Delay Timer Value		This read-only field is useful for debug purposes. It contains the value of the eight-bit Delay Timer.

TX Status Register

Figure 13-20 shows the TX Status register. Table 13-13 defines the bits in this register. Even though most of these fields are writable via DCR, this is purely for debug purposes. In normal operation, this register should not be directly written.

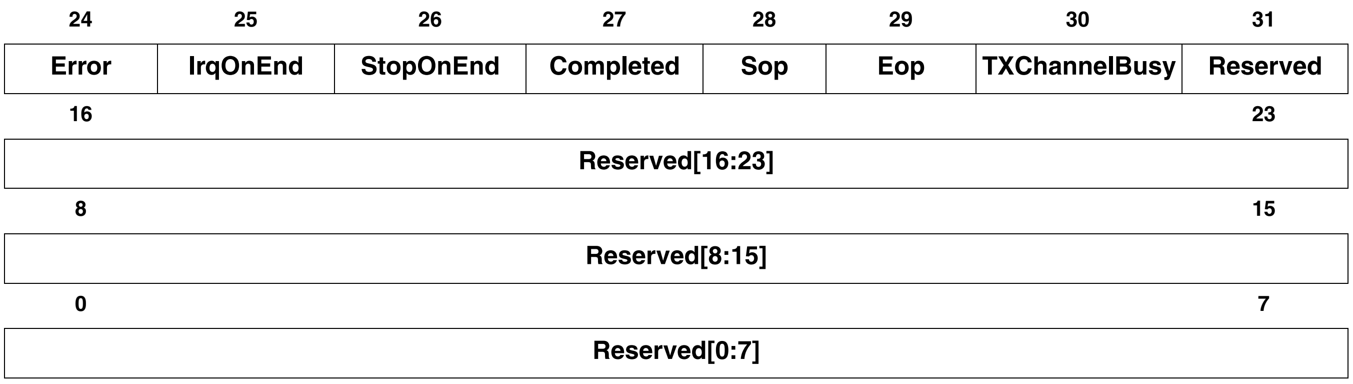


Figure 13-20: **TX Status Register**

Table 13-13: **Bit Descriptions for the TX Status Register**

Bit	Name	Default	Description
[31]	Reserved		
[30]	TX Channel Busy	`1'b0`	When cleared to zero, this bit indicates that the channel has completely flushed out all its queues and that the TX DMA has no more work allocated to it.
[29]	DMA End of Packet	`1'b0`	When set, this bit indicates that the current descriptor is the final one of a packet. For TX, the CPU sets this bit in the descriptor to indicate that this is the last descriptor of a packet to be transmitted.
[28]	DMA Start of Packet	`1'b0`	When set, this bit indicates that the current descriptor is the start of a packet. For TX, the CPU sets this bit in the descriptor to indicate that this is the first descriptor of a packet to be transmitted.

Table 13-13: **Bit Descriptions for the TX Status Register *(Cont'd)***

Bit	Name	Default	Description
[27]	DMA Completed	`1'b0`	When set, this bit indicates that the DMA has transferred all data defined by the current descriptor. In the case of TX, the DMA transfers data until the length field specified in the descriptor is zero, and then sets this bit.
[26]	DMA Stop On End	`1'b0`	When this bit is set, the DMA is forced to halt operations when the descriptor is completed. The CPU sets this bit in the status field of the descriptor. This bit is then read into the DMA TX Status register as each descriptor is processed. It is recommended that this bit be set on the EOP descriptor only.
[25]	DMA Interrupt on End	`1'b0`	When this bit is set, the DMA is forced to generate an interrupt event when the descriptor is completed. The CPU sets this bit in the status field of the descriptor. This bit is then read into the DMA TX Status register as each descriptor is processed. A typical use model would be to set this bit on the EOP descriptor only. However, it might be set for intermediate descriptors, too. Refer to the UseIntOnEnd field in the TX Channel Control register for details on how to enable this feature.
[24]	DMA Error	`1'b0`	When this bit is set, the DMA encountered a TX error. This bit is a copy of the ErrorIrq bit in the TX Interrupt register.
[0:23]	Reserved		

RX Next Descriptor Pointer

Figure 13-21 shows the RX Next Descriptor Pointer register. Table 13-14 defines the bits in this register.

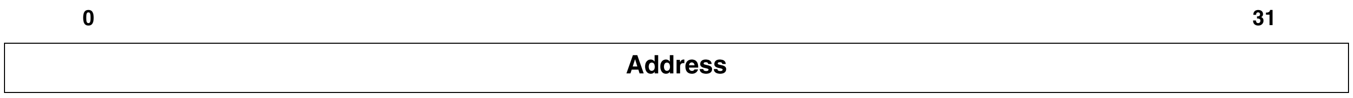


Figure 13-21: **RX Next Descriptor Pointer**

Table 13-14: **Bit Descriptions for the RX Next Descriptor Pointer**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	Contains the address of the next descriptor to be fetched. Must be eight-word aligned.

RX Current Buffer Address Register

Figure 13-22 shows the RX Current Buffer Address register. Table 13-15 defines the bits in this register.

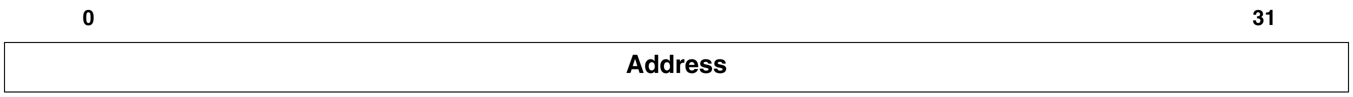


Figure 13-22: **RX Current Buffer Address Register**

Table 13-15: **Bit Descriptions for the RX Current Buffer Address Register**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	Contains the current payload address. This field changes dynamically when the DMA is operating. This address is a byte address.

RX Current Buffer Length Register

Figure 13-23 shows the RX Current Buffer Length register. Table 13-16 defines the bits in this register.

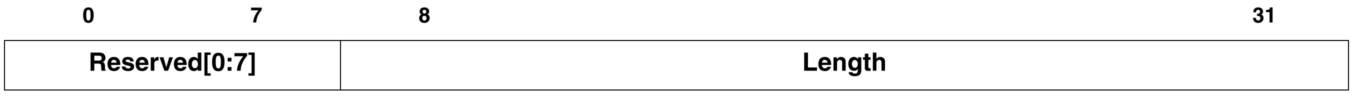


Figure 13-23: **RX Current Buffer Length Register**

Table 13-16: **Bit Descriptions for the RX Current Buffer Length Register**

Bit	Name	Default	Description
[8:31]	Length	`24'h00_0000`	Contains the remaining 24-bit payload length to be transferred. This field changes dynamically when the DMA is operating.
[0:7]	Reserved		

RX Current Descriptor Pointer

Figure 13-24 shows the RX Current Descriptor Pointer register. Table 13-17 defines the bits in this register.

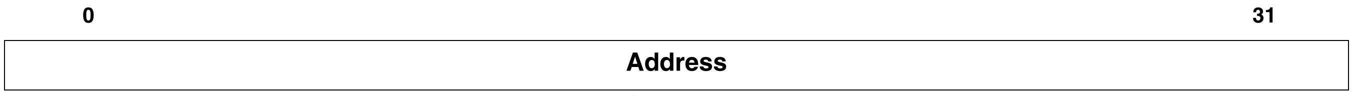


Figure 13-24: **RX Current Descriptor Pointer**

Table 13-17: **Bit Descriptions for the RX Current Descriptor Pointer**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	Contains the address of the currently executing descriptor. Must be eight-word aligned.

RX Tail Descriptor Pointer

Figure 13-25 shows the RX Tail Descriptor Pointer register. Table 13-18 defines the bits in this register.

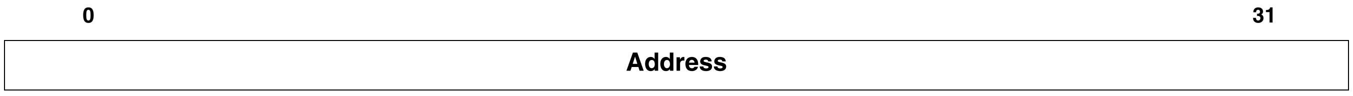


Figure 13-25: **RX Tail Descriptor Pointer**

Table 13-18: **Bit Descriptions for the RX Tail Descriptor Pointer**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	Contains the address of the last descriptor to be fetched. When this register is written to, it initiates a fetch from the address pointed to by the RX Current Descriptor Pointer register. This register can be updated dynamically, while the DMA channel is busy. The control register field, TailPtrWrEn, must be set for this feature to be enabled.

RX Channel Control Register

Figure 13-26 shows the RX Channel Control register. Table 13-19 defines the bits in this register. This register controls operation for the RX channel only. The default value of this register is user configurable through the DMAn_RXCHANNELCTRL parameters on the processor block instantiation in the user's design.

24	25 … 28	29	30	31
IrqEn	**Reserved[25:28]**	**IrqErrorEn**	**IrqDelayEn**	**IrqCoalesceEn**

16 … 19	20	21	22	23
MsbAddr[0:3]	**AppMaskEn**	**Use1BitCnt**	**UseIntOnEnd**	**LdIrqCnt**

8 … 15
IrqCount[0:7]

0 … 7
IrqTimeout[0:7]

Figure 13-26: **RX Channel Control Register**

Table 13-19: **Bit Descriptions for the RX Channel Control Register**

Bit	Name	Description
[31]	Coalescing Mechanism Interrupt Enable	Enable (1) or disable (0) the Coalescing interrupt mechanism.
[30]	Delay Timer Mechanism Interrupt Enable	Enable (1) or disable (0) the Delay Timer interrupt mechanism.
[29]	Error Detect Mechanism Interrupt Enable	Enable (1) or disable (0) the Error Detection interrupt mechanism.
[25:28]	Reserved	Reserved Bits.
[24]	Master Interrupt Enable	When this bit is set, the DMA RX channel is enabled to generate interrupts to the CPU. This is the *master* enable for the RX channel. Individual interrupt sources can be enabled or disabled separately.
[23]	Load the Interrupt Coalescing Counter	Writing a 1 to this bit forces the loading of the Interrupt Coalescing counters from the DCR IrqCount[0:7] field. This bit is self-clearing.
[22]	Use the Interrupt-On-End Mechanism	• 1: Select the interrupt-on-end mechanism for interrupt coalescing. • 0: Select the eop mechanism for interrupt coalescing.

Table 13-19: **Bit Descriptions for the RX Channel Control Register *(Cont'd)***

Bit	Name	Description
[21]	Use 1-bit Interrupt Counters	When this bit is enabled, the four-bit Interrupt Coalescing counter and two-bit Delay Timer counters are forced to be one-bit only. For certain device driver applications, this is a desirable use model.
[20]	Application Data Mask Enable	This bit enables the Application Data Mask mode. Refer to "Masking of Application Data Update," page 235 for details of operation.
[16:19]	Msb Address	These bits contain the statically assigned, most-significant four bits of the DMA address. This field must be all zeros.
[8:15]	Interrupt Coalescing Count Value	These bits contain the eight-bit value to be preloaded into the RX interrupt coalescing counter. This value is loaded into the counter when a write to the RX LdIrqCnt field is performed and subsequently reloaded whenever the Count reaches 0.
[0:7]	Interrupt Delay Time-out Value	These bits hold the compare value for the RX interrupt delay timer. The value in this register is compared to the RX Irq Delay Timer output. When they are equal, an RX interrupt event is generated.

RX Interrupt Register

This register contains the interrupt status bits for the RX channel as well as the read-only status for the RX coalescing and Delay timer counters. There are three regular interrupt sources: ErrorIrq, DelayIrq, and CoalesceIrq. There are two non-maskable interrupts (NMI): PlbRdErr and PlbWrErr. A regular interrupt can be acknowledged (and hence cleared if the corresponding counter equals 0) by writing a "1" to the respective interrupt status bit in this register. The NMIs can only be cleared by issuing a reset to the DMA (hard or soft).

Figure 13-27 shows the RX Interrupt register. Table 13-20 defines the bits in this register.

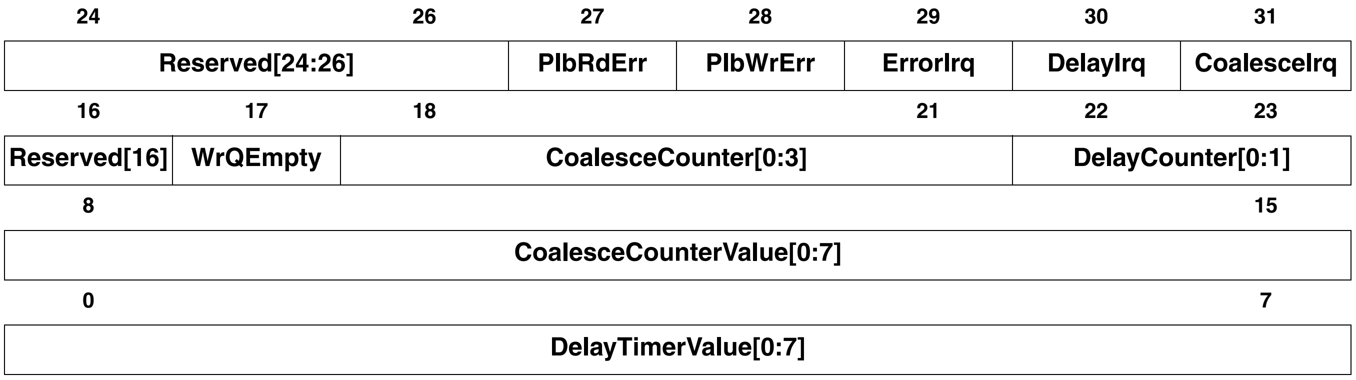


Figure 13-27: **RX Interrupt Register**

Table 13-20: **Bit Descriptions for the RX Interrupt Register**

Bit	Name	Default	Description
[31]	Coalescing Counter Interrupt	`1'b0`	When this bit is 1, the RX DMA channel has a pending interrupt because of an RX Coalescing interrupt counter greater-than-0 condition. This bit is ORed with the two other RX interrupt bits and ANDed with the RX Interrupt Enable bit to produce the RX Irq pin. Even if the RxIrqEn or IrqCoalesceEn bit is disabled, software can still poll this bit. Acknowledging an RX interrupt due to a coalescing counter condition is accomplished by writing a 1 to this bit. This action decrements the RX Coalescing interrupt counter.
[30]	Delay Timer Interrupt	`1'b0`	When this bit is 1, the RX DMA channel has a pending interrupt because of an RX Delay Timer interrupt counter greater-than-0 condition. This bit is ORed with the two other RX interrupt bits and ANDed with the RX Interrupt Enable bit to produce the RX Irq pin. Even if the RxIrqEn or IrqDelayEn bit is disabled, software can still poll this bit. Acknowledging an RX interrupt due to a Delay Timer counter condition is accomplished by writing a 1 to this bit. This action decrements the RX Delay Timer interrupt counter.
[29]	Error Interrupt	`1'b0`	When this bit is 1, the RX DMA channel has a pending interrupt because of an RX error that has occurred. This bit is ORed with the two other RX interrupt bits and ANDed with the RX Interrupt Enable bit to produce the RX Irq pin. Even if the RxIrqEn or IrqErrorEn bit is disabled, software can still poll this bit. Acknowledging an RX interrupt due to an Error is accomplished by writing a 1 to this bit. This action clears this bit.
[28]	PLB Write Error Non-Maskable Interrupt	`1'b0`	When this bit is 1, the RX DMA channel has received an error from the PLB due to a PLB write operation. This serious error causes the DMA to freeze the LocalLink interface as soon as it receives this indication from the crossbar. This bit can only be cleared by resetting the DMA (hard or soft).
[27]	PLB Read Error Non-Maskable Interrupt	`1'b0`	When this bit is 1, the RX DMA channel has received an error from the PLB due to a PLB read operation. This serious error causes the DMA to freeze the LocalLink interface as soon as it receives this indication from the crossbar. This bit can only be cleared by resetting the DMA (hard or soft).
[24:26]	Reserved		
[22:23]	Delay Timer Interrupt Counter		This read-only field is useful for debug purposes. It contains the value of the two-bit counter used to store the number of RX Delay Timer interrupts that are outstanding.
[18:21]	Coalescing Interrupt Counter		This read-only field is useful for debug purposes. It contains the value of the four-bit counter used to store the number of RX coalescing counter interrupts that are outstanding.
[17]	Write Command Queue Empty Status		This read-only field is useful for debug purposes. It indicates whether the Write Command Queue is empty (1) or not (0). If the DMA is paused, reading this field indicates that all the write data associated with the pending commands has been flushed.

Table 13-20: **Bit Descriptions for the RX Interrupt Register *(Cont'd)***

Bit	Name	Default	Description
[16:]	Reserved		
[8:15]	Coalescing Counter Value		This read-only field is useful for debug purposes. It contains the value of the eight-bit Coalescing Counter.
[0:7]	Delay Timer Value		This read-only field is useful for debug purposes. It contains the value of the eight-bit Delay Timer.

RX Status Register

Figure 13-28 shows the RX Status register. Table 13-21 defines the bits in this register. Even though most of these fields are writable via DCR, this register is purely for debug purposes. In normal operation, this register should not be directly written.

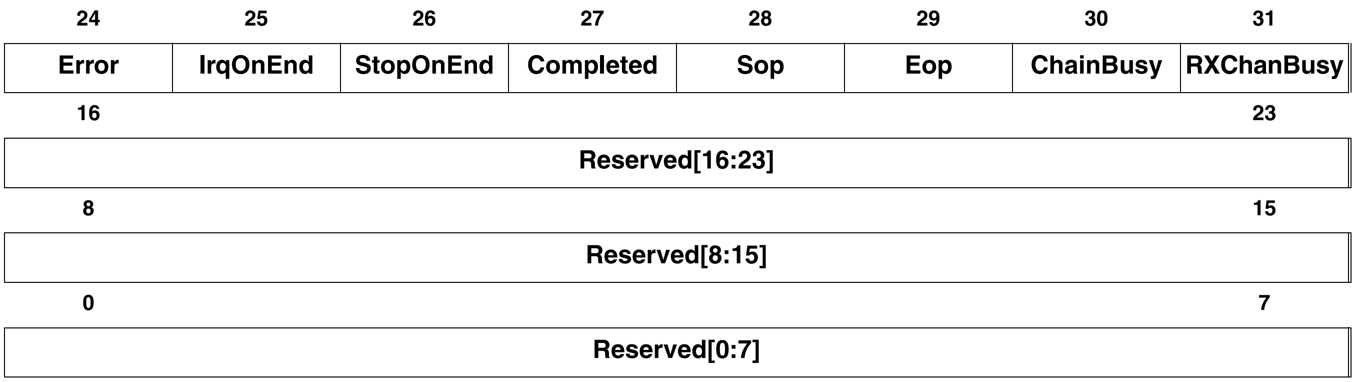


Figure 13-28: **RX Status Register**

Table 13-21: **Bit Descriptions for the RX Status Register**

Bit	Name	Default	Description
[31]	RX Channel Busy	`1'b0`	When this bit is cleared to zero, the channel has completely flushed out all its queues AND the RX DMA has no more work allocated to it.
[30]	DMA Descriptor Chain Busy	`1'b0`	When this bit is set, the Descriptor Chain is still active/busy. This means the DMA still has pending descriptors to process. When this bit is cleared, it DOES NOT guarantee that all the queues have been flushed out to memory; there might still be some pending descriptor writes.
[29]	DMA End of Packet	`1'b0`	When this bit is set, the current descriptor is the final one of a packet. For RX, when an EOP is received by the LocalLink interface, the DMA sets this bit in the descriptor to inform the CPU that the current descriptor is the last of a received packet.
[28]	DMA Start of Packet	`1'b0`	When this bit is set, the current descriptor is the start of a packet. For RX, when an SOP is received by the LocalLink interface, the DMA sets this bit in the descriptor to inform the CPU that the current descriptor is the first of a received packet.

Table 13-21: **Bit Descriptions for the RX Status Register *(Cont'd)***

Bit	Name	Default	Description
[27]	DMA Completed	`1'b0`	When this bit is set, the DMA has transferred all data defined by the current descriptor. For RX, the DMA transfers data until the length field specified in the descriptor is zero OR when it receives an EOP indication from the LocalLink interface. At that point, the DMA sets this bit.
[26]	DMA Stop On End	`1'b0`	When this bit is set, the DMA is forced to halt operations when the descriptor is completed. The CPU sets this bit in the status field of the descriptor. This bit is then read into the DMA RX Status register as each descriptor is processed. It is recommended that this bit be set, corresponding to an EOP descriptor only.
[25]	DMA Interrupt on End	`1'b0`	When this bit is set, the DMA is forced to generate an interrupt event when the descriptor is completed. The CPU sets this bit in the status field of the descriptor. This bit is then read into the DMA RX Status register as each descriptor is processed. A typical use model is to set this bit on the EOP descriptor only. However, it can be set for intermediate descriptors, too. Refer to the UseIntOnEnd field in the RX Channel Control register for details of how to enable this feature.
[24]	DMA Error	`1'b0`	When this bit is set, the DMA encountered an RX error. This bit is a copy of the ErrorIrq bit in the RX Interrupt register.
[0:23]	Reserved		

DMA Control Register

Figure 13-29 shows the DMA Control register. Table 13-22 defines the bits in this register. This register contains control fields that affect both the RX and TX channels. The default value of bit 31 is 0. The default value of bits 26:29 is determined by bits 2:5 of the DMAn_CONTROL attribute on the processor block instantiation in the user's design.

24	25	26	27	28	29	30	31
Reserved[24:25]		**PlbErrDisable**	**OverFlowErrDisable[0:1]**		**TailPtrEn**	**Reserved**	**SwReset**

0	23
Reserved[0:23]	

Figure 13-29: **DMA Control Register**

Table 13-22: **Bit Descriptions for the DMA Control Register**

Bit	Name	Description
[31]	Software Reset	Writing a 1 to this bit forces the DMA engine (both RX and TX channels) to shut down and reset itself. Because the DMALLRSTENGINEACK output is asserted when this bit is a 1, it can be used to reset a remote LocalLink device while the DMA engine is resetting itself. After setting this bit, software must poll it until the bit is cleared by the DMA, which indicates that the reset process is done and the pipeline has been flushed.
[30]	Reserved	

Table 13-22: **Bit Descriptions for the DMA Control Register *(Cont'd)***

Bit	Name	Description
[29]	Tail Pointer Enable	When this bit is set, the Tail Pointer mechanism is enabled. In this mode, writing to the tail pointer initiates a DMA transaction and the comparison (tail pointer == current pointer) ends descriptor execution. When cleared, the legacy mode of writing to the current pointer to initiate a transfer is supported. Refer to "DMA Legacy Mode," page 231 for details.
[27:28]	Overflow Counter Error Interrupt Disable	When this bit is set, the error interrupt is disabled when either the two-bit Delay Timer counter or the four-bit Coalescing counter overflows. Bit [27] is used for the RX channel, and Bit [28] is used for the TX channel.
[26]	PLB Error Disable	When this bit is set, error checking is disabled due to reads/writes to and from the crossbar PLB. If one of these errors occurs, the DMA reacts as follows: • PLB Error Disable = `1'b1`: The DMA ignores the error and continues as usual. • PLB Error Disable = `1'b0`: • Read Data Error. The DMA logs a plb_rd_error NMI bit in the appropriate interrupt register (RX or TX). The LocalLink interface is frozen immediately. • Write Data Error. The DMA logs a plb_wr_error NMI bit in both RX and TX interrupt registers. The LocalLink interfaces is frozen immediately.
[0:25]	Reserved	

Physical Interface

Each DMA engine has the following interfaces:

- LocalLink interface
- Miscellaneous

Table 13-23: **DMA Controller Signals**

Signal Name	Dir	Description
LocalLink Interface		
DMALLTXD[0:31]	Out	This 32-bit bus contains the TX data. It is valid when DMALLTXSRCRDYN and LLDMATXDSTRDYN are asserted.
DMALLTXREM[0:3]	Out	The TX remainder bus is used as the data mask for the last word of the header, the payload, or the footer.
DMALLTXSOFN	Out	This active-Low signal is asserted to indicate the TX start of frame.
DMALLTXEOFN	Out	This active-Low signal is asserted to indicate the TX end of frame.
DMALLTXSOPN	Out	This active-Low signal is asserted to indicate the TX start of payload.
DMALLTXEOPN	Out	This active-Low signal is asserted to indicate the TX end of payload.
DMALLTXSRCRDYN	Out	This active-Low signal is asserted to indicate the TX source is ready and the DMA has valid data on outputs.
DMALLRXDSTRDYN	Out	This active-Low signal is asserted to indicate the RX destination is ready and the DMA is ready to receive data.

Table 13-23: **DMA Controller Signals *(Cont'd)***

Signal Name	Dir	Description
DMALLRSTENGINEACK	Out	This active-High signal is asserted to acknowledge a DMA engine reset. It is asserted as soon as the SwReset bit is written or the DMALLRSTENGINEREQ signal is asserted. This signal is deasserted when the DMA has completed its internal reset sequence. This pin can be used to reset external devices.
LLDMARSTENGINEREQ	In	This active-High signal is asserted to request a DMA engine reset. This signal should be asserted for one cycle only when a reset is required. The reset is completed when the DMALLRSTENGINEACK signal is Low.
LLDMATXDSTRDYN	In	This active-Low signal is asserted to indicate the TX connecting device is ready to receive data.
LLDMARXD[0:31]	In	This 32-bit bus contains RX data. It is valid when DMALLRXSRCRDY and LLDMARXDSTRDY are asserted.
LLDMARXREM[0:3]	In	The RX remainder bus is used as the data mask for the last word of the header, the payload, or the footer.
LLDMARXSOFN	In	This active-Low signal is asserted to indicate the RX start of frame.
LLDMARXEOFN	In	This active-Low signal is asserted to indicate the RX end of frame.
LLDMARXSOPN	In	This active-Low signal is asserted to indicate the RX start of payload.
LLDMARXEOPN	In	This active-Low signal is asserted to indicate the RX end of payload.
LLDMARXSRCRDYN	In	This active-Low signal is asserted to indicate the RX connecting device has valid data on the RX LocalLink outputs.
CPMDMALLCLK	In	This input provides the clock for the LocalLink interface and DMA.
Miscellaneous Signals		
DMATXIRQ	Out	This output is the DMA engine TX interrupt to the processor.
DMARXIRQ	Out	This output is the DMA engine RX interrupt to the processor.

Section IV: Programming Considerations

Chapter 14

DCR Programming Considerations

Overview of the Device Control Registers (DCRs) Map

DCR registers are CPU accessible registers for device configuration, control, and statuses. The Processor Block DCR map is shown in Table 14-1. The map contains a block of 256 locations that is relocatable with the starting address defined by a two-bit tie-off value, **TIEDCRBASEADDR**[0:1][1].

Table 14-1: **Processor Block DCR Map**

Block	Sub-Block	Address Offset and Range
DCR Controller	DCR Controller	`0x00 – 0x02`
	Reserved	`0x03`
APU Controller	Auxiliary Processor Unit (APU) Controller	`0x04 – 0x05`
	Reserved	`0x06 – 0x0F`
MCI	Memory Controller Interface (MCI)	`0x10 – 0x12`
	Reserved	`0x13 – 0x1F`
PLB Interfaces + Crossbar	Crossbar	`0x20 – 0x33`
	PLB Slave 0 (PLBS0)	`0x34 – 0x43`
	PLB Slave 1 (PLBS1)	`0x44 – 0x53`
	PLB Master (PLBM)	`0x54 – 0x5F`
Reserved		`0x60 – 0x7F`
DMA Engines	DMA Engine 0 (DMAC0)	`0x80 – 0x90`
	Reserved	`0x91 – 0x97`
	DMA Engine 1 (DMAC1)	`0x98 – 0xA8`
	Reserved	`0xA9 – 0xAF`
	DMA Engine 2 (DMAC2)	`0xB0 – 0xC0`
	Reserved	`0xC1 – 0xC7`
	DMA Engine 3 (DMAC3)	`0xC8 – 0xD8`
	Reserved	`0xD9 – 0xDF`
Reserved		`0xE0 – 0xFF`

1. In this document, bold upper case names are software names.

Detailed Descriptions

The register map can be divided into five sections corresponding to the five main functional blocks (excluding the processor) within the embedded processor block in Virtex-5 FXT FPGAs:

- DCR controller
- Auxiliary Processing Unit (APU) Controller
- Memory Controller Interface (MCI)
- PLB Interfaces and Crossbar
- DMA Engines

Detailed descriptions of the DCR registers are provided in the following sections. The same information is also available from individual design specifications.

DCR Controller (0x00 – 0x02)

There are three registers in the DCR controller. These registers are needed for indirect addressing, arbitration, and interface mode select.

Register 0x00: Indirect Address Register

This register contains the address used in indirect addressing. The indirect address is formed by a 2-bit upper address bus (UABUS[20:21]) value and a 10-bit address bus (ABUS[0:9]) value. This register, shown in Figure 14-1, is both readable and writable. All unused bits in the register return 0s when read.

0 – 19	20 – 21	22 – 31
Reserved	UABUS[20:21]	ABUS[0:9]

Figure 14-1: **Register 0x00**

Register 0x01: Indirect Access Register

This location is used as a proxy to indirectly access the DCR slaves. When location `0x01` is accessed, the DCR controller replaces the DCR address (`0x01`) with the content of register `0x00` for address decoding. The DCR master reads or writes to the 12-bit address stored in register `0x00`. This location is both readable and writable.

Register 0x02: Control, Configuration, and Status Register

Register `0x02`, shown in Figure 14-2, handles control, configuration, and status. Table 14-2 describes the fields within the register.

0	1	2	3	4	5	6	7	8 – 29	30	31
c440 lock	c440 alock	xm lock	xm alock	auto-lock	xm asyn	xs asyn	xm towait	Reserved	c440 time out	xm time out

Figure 14-2: **Register 0x02**

Table 14-2: **Bit Descriptions for Register 0x02**

Bit	Name	Dir	Default	Description
0	c440 lock	R/W	0	Processor bus lock bit. Can be written to and read by the processor DCR master. The external master can also read this bit.
1	c440 alock	RO	0	Processor auto bus lock bit.
2	xm lock	R/W	0	External master bus lock bit. Can be written to and read by the external DCR master. The processor DCR master can also read this bit.
3	xm alock	RO	0	External DCR master auto bus lock bit.
4	auto-lock	R/W	1	Configures the auto-lock feature. The default value for this bit is 1 to enable the auto-lock. This bit is cleared to disable the auto-lock function. This bit is initialized by the embedded processor block attribute DCR_AUTOLOCK_ENABLE.
5	xm asyn	RO	0	Indicates the external DCR master interface asynchronous mode. • 0: Synchronous mode • 1: Asynchronous mode This bit is initialized by the embedded processor block attribute PPCDM_ASYNCMODE.
6	xs asyn	RO	0	Indicates the external DCR slave interface asynchronous mode. • 0: Synchronous mode • 1: Asynchronous mode This bit is initialized by the embedded processor block attribute PPCDS_ASYNCMODE.
7	xm towait	R/W	0	Configures the external DCR master time-out wait support. By default, this bit is 0, so that the external DCR master is assumed not to support time-out waits (the signal is tied to 0), but this setting also works with a master that supports time-out waits. This bit is set to 1 if the external master supports time-out waits, allowing for better performance for the external master if the processor DCR Master locks the bus.
8:29	Reserved	-	0	Reserved.
30	c440 timeout	Read/ Clear	0	Set if a processor DCR master access time-out occurs. This bit is cleared on writes. If the bus is locked, only the locking master can clear it, and the other master can read it but not clear it.
31	xm timeout	Read/ Clear	0	Set if an external DCR master access time-out occurs. This bit is cleared on writes. If the bus is locked, only the locking master can clear it, and the other master can read it but not clear it.

The DCR controller prevents more than one master from locking the bus, so writing to bit 0 or 2 might not lead to changes in those bit locations. All unused bits in this register return 0s when read.

APU Controller (0x04 – 0x05)

Register 0x04: User-Defined Instruction (UDI) Configuration Registers

For all UDIs that are not loads or stores, the user needs to configure the primary and extended opcodes along with any necessary execution options. Figure 14-3 shows the UDI Configuration register bits. Table 14-3 defines the bits in the UDI Configuration register. The UDI configuration registers are initialized by embedded processor block attributes APU_UDI0 through APU_UDI15.

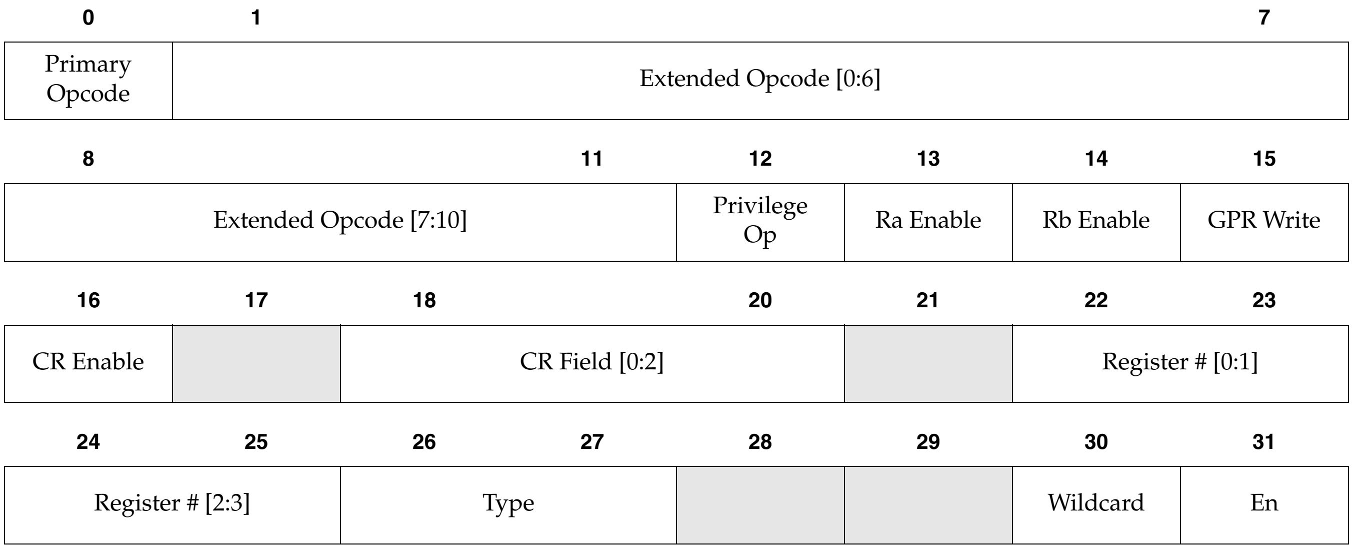


Figure 14-3: **UDI Configuration Register**

Table 14-3: **Bit Descriptions for the UDI Configuration Register**

Bit	Name	Description
0	Primary Opcode	• 0: `6'b000000` (opcode 0) • 1: `6'b000100` (opcode 4)
1:11	Extended Opcode	11 bits of the full extended opcode
12	Privilege Op	When this bit is set, this instruction must execute in privilege mode.
13	Ra Enable	When this bit is set, this instruction needs to read the Ra source operand from the GPR.
14	Rb Enable	When this bit is set, this instruction needs to read the Rb source operand from the GPR.
15	GPR Write	When this bit is set, this instruction writes a result to the Rt register in the GPR.
16	CR Enable	When this bit is set, this instruction returns Condition Record (CR) bits to the CR field indicated in CRField[0:2].
17	Reserved	Reserved
18:20	CRField[0:2]	Indicates which field receives the condition record (if the CR Enable bit in the instruction is set to 1).

Table 14-3: **Bit Descriptions for the UDI Configuration Register *(Cont'd)***

Bit	Name	Description
21	Reserved	Reserved
22:25	Register #	Indicates the register number (0 – 15). This value is only used when setting the DCR read/write pointer (Type = `2'b11`), but can be viewed when reading the UDI contents through a DCR read.
26:27	Type	Indicates the operation type of the instruction: • `00`: Non-autonomous, early confirm • `01`: Non-autonomous, late confirm • `10`: Autonomous • `11`: Sets read/write pointer to value in Register # field
28:29	Reserved	Reserved
30	Wildcard	When this bit is set, bits [1:5] are not considered as the extended opcode but can be anything. Instead only bits [6:11] are checked. When this bit is cleared, the entire 11 bits of the extended opcode are checked.
31	En	This bit enables the UDI. It indicates that the opcode and options written in the UDI are valid and should be used during decode.

Register 0x05: APU Control Register

The APU Control register turns on or off various features in the APU controller. Figure 14-4 shows the bits in the APU Control register. Table 14-4 defines the bits within the register. The APU control register is initialized by embedded processor block attribute APU_CONTROL.

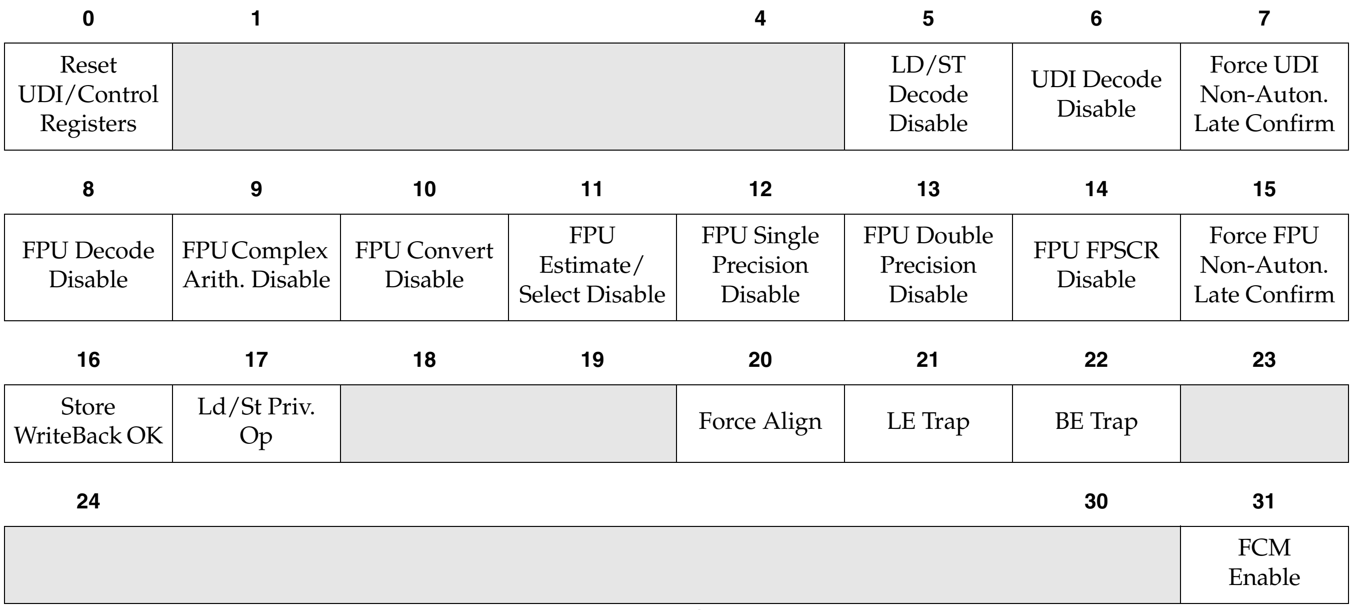

0
1
4
5
6
7
Reset UDI/Control Registers
LD/ST Decode Disable
UDI Decode Disable
Force UDI Non-Auton. Late Confirm
8
9
10
11
12
13
14
15
FPU Decode Disable
FPU Complex Arith. Disable
FPU Convert Disable
FPU Estimate/ Select Disable
FPU Single Precision Disable
FPU Double Precision Disable
FPU FPSCR Disable
Force FPU Non-Auton. Late Confirm
16
17
18
19
20
21
22
23
Store WriteBack OK
Ld/St Priv. Op
Force Align
LE Trap
BE Trap
24
30
31
FCM Enable

Figure 14-4: **APU Control Register**

Table 14-4: **Bit Descriptions for the APU Control Register**

Bit	Name	Default	Description
0	Reset UDI/Control Registers	-	When a 1 is written to this bit, all the UDI registers are reset to their default values. The rest of the bits in the control register are also reset to their default values. When read, this bit always returns a 0.
1:4	Reserved	-	Reserved
5	LD/ST Decode Disable	0	When set, this bit disables all FCM Load/Store decoding in the APU controller. This does not affect FPU Load/Store instructions. An FCM Load/Store in the program causes an illegal instruction exception.
6	UDI Decode Disable	0	When set, this bit disables all UDI decoding in the APU controller. This does not affect FCM Load/Store or FPU instructions. A UDI instruction in the program causes an illegal instruction exception.
7	Force UDI Non-Autonomous, Late Confirm	0	When set, this bit forces any non-storage UDI instruction to be executed as a non-autonomous instruction with late confirm regardless of the type indicated in the UDI register.
8	FPU Decode Disable	1	When set, this bit disables all FPU decoding in the APU controller. An FPU instruction in the program causes an illegal instruction exception.
9	FPU Complex Arithmetic Disable	0	When set, this bit disables decoding for all FPU divide and square root instructions (fdiv, fdiv., fdivs, fdivs., fsqrt, fsqrt., fsqrts, fsqrts.). An FPU complex arithmetic instruction in the program when FPU Decode is not disabled causes an unimplemented instruction exception.
10	FPU Convert Disable	0	When set, this bit disables decoding for all FPU convert instructions (fcfid, fctid, fctidz, fctiw, fctiw., fctiwz, fctiwz., frsp, frsp.). An FPU convert instruction in the program when FPU Decode is not disabled causes an unimplemented instruction exception.
11	FPU Estimate/select Disable	0	When set, this bit disables decoding for all FPU estimate instructions (fres, fres., frsqrte, frsqrte., fsel, fsel.). An FPU estimate instruction in the program when FPU Decode is not disabled causes an unimplemented instruction exception.
12	FPU Single Precision Disable	0	When set, this bit disables decoding for all FPU single-precision only instructions (lfs, lfsu, lfsx, lfsux, stfs, stfsu, stfsx, stfsux, fadds, fadds., fsubs, fsubs., fdivs, fdivs., fmuls, fmuls., fsqrts, fsqrts., fmadds, fmadds., fnmadds, fnmadds., fmsubs, fmsubs., fnmsubs, fnmsubs.). A single-precision FPU instruction in the program when FPU Decode is not disabled causes an unimplemented instruction exception.
13	FPU Double Precision Disable	0	When set, this bit disables decoding for all FPU double-precision only instructions (lfd, lfdu, lfdx, lfdux, stfd, stfdu, stfdx, stfdux, stfiwx, fadd, fadd., fsub, fsub., fdiv, fdiv., fmul, fmul., fsqrt, fsqrt., fmadd, fmadd., fnmadd, fnmadd., fmsub, fmsub., fnmsub, fnmsub.). If a double-precision FPU instruction is encountered in the program and FPU Decode is not disabled, an unimplemented instruction exception occurs.
14	FPU FPSCR Disable	0	When set, this bit disables decoding for all FPSCR FPU instructions (mcrfs, mffs, mffs., mtfsb0, mtfsb0., mtfsb1, mtfsb1., mtfsf, mtfsf., mtfsfi, mtfsfi.). An FPSCR instruction in the program when FPU Decode is not disabled causes an unimplemented instruction exception.

Table 14-4: **Bit Descriptions for the APU Control Register *(Cont'd)***

Bit	Name	Default	Description
15	Force FPU Non-Autonomous, Late Confirm	0	When set, this bit forces all non-storage FPU instructions to be executed as non-autonomous instructions with late confirm.
16	Store WritebackOK	0	When this bit is set, the APU controller waits to send a WritebackOK signal to the FCM for all store instructions (both APU and FPU stores). The WritebackOK signal is sent after the store instruction passes the LWB stage in the CPU pipe, which can cause a slight performance hit when executing store instructions.
17	LD/ST Privilege	0	When this bit is set, any load or store UDI executes in privileged mode (this does not affect FPU load/store instructions).
18:19	Reserved	-	Reserved
20	Force Align	0	When this bit is set, any load or store (both APU and FPU) forces alignment. The address is forced to align on the natural boundary of the transfer (word boundary for a word transfer, doubleword boundary for a doubleword transfer, and so forth). This also prevents an alignment exception.
21	LE Trap	0	When this bit is set, any load or store (both APU and FPU) traps when the Endian storage attribute is `1'b1` (little Endian).
22	BE Trap	0	When this bit is set, any load or store (both APU and FPU) traps when the Endian storage attribute is `1'b0` (big Endian).
23:30	Reserved	-	Reserved
31	FCM Enable	0	When this bit is set, the FCM interface is enabled and the APU controller decodes instructions. When this bit is cleared, bits 5, 6, and 8 are overridden. The APU controller does not decode any instructions.

Memory Controller Interface (0x10 – 0x12)

Register 0x10 – MI_CONTROL [0:31]

This register is initialized by embedded processor block attribute MI_CONTROL.

Table 14-5: **Bit Descriptions for the MI_CONTROL Register**

Bit	Name	Default	Description
[0]	enable	0	• 1: The MCI is enabled, and PLB and DMA masters can access the soft memory controller through the crossbar. • 0: The MCI is disabled, and any attempt to access the MCI through the crossbar will fail.
[1]	Rowconflictholdenable	0	If there is a change between the row from the current address and the past address, setting this bit causes the MCI block to wait Autoholdduration number of cycles before starting up the next instruction, assuming the MCMIADDRREADYTOACCEPT signal is asserted.

Table 14-5: **Bit Descriptions for the MI_CONTROL Register *(Cont'd)***

Bit	Name	Default	Description
[2]	Bankconflictholdenable	0	If there is a change between the bank from the current address and the past address, setting this bit causes the MCI block to wait Autoholdduration number of cycles before starting up the next instruction, assuming the MCMIADDRREADYTOACCEPT signal is asserted.
[3]	Directionconflictholdenable	0	If there is a change of direction between the current address and the past address (from reads to writes and writes to reads), setting this bit causes the MCI block to wait Autoholdduration number of cycles before starting up the next instruction, assuming the MCMIADDRREADYTOACCEPT signal is asserted.
[4:5]	Autoholdduration	00	This field tells the MCI block how long to hold off, while waiting for the memory controller's MCMIADDRREADYTOACCEPT signal to become asserted, assuming there was a triggering event causing an autohold. • `00`: 2 cycles • `01`: 3 cycles • `10`: 4 cycles • `11`: 5 cycles
[6]	2:3 Clock Ratio mode	0	Clock ratio mode: • `0`: Integer ratio of the MCI clock to the embedded processor block interconnect clock (CPMINTERCONNECTCLK) • `1`: Fractional ratio of the MCI clock to the embedded processor block interconnect clock (CPMINTERCONNECTCLK) (3/2)
[7]	overlaprdwr	0	In DDR mode, after every read or write, the equivalent amount of time of the burst is inserted before requesting the next transaction. In QDR mode, the reads and writes are separate transactions. In this mode, a read transaction does not block the next write transaction from going out.
[8:9]	Burstwidth	00	Data per clock cycle: • `00`: Burst width = 128 • `01`: Burst width = 64 • `10`: Reserved • `11`: Burst width = 32
[10:11]	Burstlength	00	Burst length: • `00`: Burst length = 1 • `01`: Burst length = 2 • `10`: Burst length = 4 • `11`: Burst length = 8
[12:15]	Write Data Delay (WDD)	0000	Values 0 through 10 are defined. Values 11 through 15 are reserved.
16	RMW	0	Allows the MCI block to always autohold if all the byte enables for a write are not turned on for that transaction.
[17:23]	Reserved	0000000	These bits are reserved

Table 14-5: **Bit Descriptions for the MI_CONTROL Register *(Cont'd)***

Bit	Name	Default	Description
24	PLB Priority Enable	`1`	• `0`: First level arbitration is disabled for the PLB Masters trying to access the MCI through the crossbar. • `1`: First level arbitration is enabled among the PLB Masters trying to access the MCI through the crossbar See "Arbitration" in Chapter 3 for more information.
[25:27]	Reserved	`000`	These bits are reserved
[28]	Pipelined Read Enable	`1`	• `0`: The crossbar does not accept a new read command until the current read command completes. • `1`: The crossbar accepts read commands destined for the MCI while the current read operation is still in progress.
[29]	Pipelined Write Enable	`1`	• `0`: The crossbar does not accept a new write command until the current write command completes. • `1`: The crossbar accepts write commands destined for the MCI while the current write operation is still in progress.
[30]	Reserved	`1`	Reserved
[31]	Reserved	`1`	Reserved

Register 0x11: MI_ROWCONFLICT_MASK [0:31]

This register contains the mask used to detect row conflicts from one transaction to another. This register is at DCR address `0x11`. The 32 bits in this register correspond to the higher order 32 bits of the 36-bit address generated by the MCI. A 1 in any bit position identifies that bit as a row address bit. For example, if bits 8:20 are set to 1, the MIMCROWCONFLICT signal is set to 1 if the corresponding bits of MIMCADDRESS change between the previous instruction sent to the soft memory controller and this instruction. The default value of this register is 0.

Register 0x12: MI_BANKCONFLICT_MASK [0:31]

This register contains the mask used to detect bank conflicts from one transaction to another. This register is at DCR address `0x12`. The 32 bits in this register correspond to the higher order 32 bits of the 36-bit address generated by the MCI. A 1 in any bit position identifies that bit as a bank address bit. For example, if bits 4:7 are set to 1, the MIMCBANKCONFLICT signal is set to 1 if the corresponding bits of MIMCADDRESS change between the previous instruction sent to the soft memory controller and this instruction. The default value of this register is 0.

DCRs for the PLB Interfaces and Crossbar (0x20 – 0x5F)

A block of 64 DCR locations (`0x20` to `0x5F`) is allocated for use by the crossbar, two PLB slaves (PLBS0 and PLBS1), the PLB Master (MPLB), and the Address Map configuration registers. Four separate DCR lists are shown in Table 14-6 through Table 14-9.

All the interrupt status bits of the PLB interfaces and the crossbar are consolidated in the Interrupt Status register at `0x20`.

There are 54 registers defined in Table 14-6 to Table 14-9. Twenty registers (including 13 for address mapping) use tie-off values (embedded processor block attributes) rather than hardware-defined default values after reset. The tie-offs allow user control over the register default values.

Table 14-6: **List of DCRs for the Crossbar**

Address	Mnemonic	Description	Type
Global Configuration and Status			
`0x20`	IST	Interrupt Status Register	Clear on Write to bit and Read Only
`0x21`	IMASK	Interrupt Mask Register	R/W
`0x22`	-	Reserved	-
Crossbar for PLB Master Configuration			
`0x23`	ARB_XBC	Arbitration Configuration Register	R/W
Crossbar for PLB Master Status			
`0x24`	FIFOST_XBC	FIFO Overflow and Underflow Status	Clear on Write to bit
Crossbar for PLB Master Hardware Debug			
`0x25`	-	Reserved	-
`0x26`	MISC_XBC	Miscellaneous Control and Status	R/W, Write Only
`0x27`	-	Reserved	-
Crossbar for MCI Configuration			
`0x28`	ARB_XBM	Arbitration Configuration Register	R/W
Crossbar for MCI Status			
`0x29`	FIFOST_XBM	FIFO Overflow and Underflow Status	Clear on Write to bit
Crossbar for MCI Hardware Debug			
`0x2A`	SM_ST_XBM	State Machine States Register	Read Only
`0x2B`	MISC_XBM	Miscellaneous Control and Status	R/W, Write Only
`0x2C`	-	Reserved	-
Address Map Configuration			
`0x2D`	TMPL0_XBAR_MAP	Template Register 0 for Crossbar	R/W
`0x2E`	TMPL1_XBAR_MAP	Template Register 1 for Crossbar	R/W
`0x2F`	TMPL2_XBAR_MAP	Template Register 2 for Crossbar	R/W
`0x30`	TMPL3_XBAR_MAP	Template Register 3 for Crossbar	R/W
`0x31`	TMPL_SEL_REG	Template Selection Register	R/W
`0x32`	-	Reserved	-
`0x33`	-	Reserved	-

Table 14-7: **List of DCRs for PLB Slave 0 (SPLB 0)**

Address	Mnemonic	Description	Type
Configuration			
0x34	CFG_PLBS0	Configuration Register	R/W
0x35	-	Reserved	-
Status			
0x36	SEAR_U_PLBS0	Slave Error Address Register, upper 4 bits	Clear on Write to 0x38
0x37	SEAR_L_PLBS0	Slave Error Address Register, lower 32 bits	Clear on Write to 0x38
0x38	SESR_PLBS0	Slave Error Status Register	Clear on Write
0x39	MISC_ST_PLBS0	Miscellaneous Status Register	Clear on Write to bit
0x3A	PLBERR_ST_PLBS0	PLB Error Status	Clear on Write to bit
Hardware Debug			
0x3B	SM_ST_PLBS0	State Machine States Register	Read Only
0x3C	MISC_PLBS0	Miscellaneous Control and Status	R/W, WO, RO
0x3D	CMD_SNIFF_PLBS0	Command Sniffer	R/W
0x3E	CMD_SNIFFA_PLBS0	Command Sniffer Address	R/W
0x3F	-	Reserved	-
Address Map			
0x40	TMPL0_PLBS0_MAP	Template Register 0	R/W
0x41	TMPL1_PLBS0_MAP	Template Register 1	R/W
0x42	TMPL2_PLBS0_MAP	Template Register 2	R/W
0x43	TMPL3_PLBS0_MAP	Template Register 3	R/W

Table 14-8: **List of DCRs for the PLB Slave 1 (SPLB 1)**

Address	Mnemonic	Description	Type
Configuration			
0x44	CFG_PLBS1	Configuration Register	R/W
0x45	-	Reserved	-
Status			
0x46	SEAR_U_PLBS1	Slave Error Address Register, upper 4 bits	Clear on Write to 0x48
0x47	SEAR_L_PLBS1	Slave Error Address Register, lower 32 bits	Clear on Write to 0x48
0x48	SESR_PLBS1	Slave Error Status Register	Clear on Write
0x49	MISC_ST_PLBS1	Miscellaneous Status Register	Clear on Write to bit
0x4A	PLBERR_ST_PLBS1	PLB Error Status	Clear on Write to bit

Table 14-8: **List of DCRs for the PLB Slave 1 (SPLB 1)** ***(Cont'd)***

Address	Mnemonic	Description	Type
Hardware Debug			
`0x4B`	SM_ST_PLBS1	State Machine States Register	Read Only
`0x4C`	MISC_PLBS1	Miscellaneous Control and Status	R/W, WO, RO
`0x4D`	CMD_SNIFF_PLBS1	Command Sniffer	R/W
`0x4E`	CMD_SNIFFA_PLBS1	Command Sniffer Address	R/W
`0x4F`	-	Reserved	-
Address Map			
`0x50`	TMPL0_PLBS1_MAP	Template Register 0	R/W
`0x51`	TMPL1_PLBS1_MAP	Template Register 1	R/W
`0x52`	TMPL2_PLBS1_MAP	Template Register 2	R/W
`0x53`	TMPL3_PLBS1_MAP	Template Register 3	R/W

Table 14-9: **List of DCRs for the PLB Master (MPLB)**

Address	Mnemonic	Description	Type
Configuration			
`0x54`	CFG_PLBM	Configuration Register	R/W
`0x55`	-	Reserved	-
Status			
`0x56`	FSEAR_U_PLBM	FPGA Logic Slave Error Address Register, upper 4 bits	Clear on Write to `0x58`
`0x57`	FSEAR_L_PLBM	FPGA Logic Slave Error Address Register, lower 32 bits	Clear on Write to `0x58`
`0x58`	FSESR_PLBM	FPGA Logic Slave Error Status Register	Clear on Write
`0x59`	MISC_ST_PLBM	Miscellaneous Status	Clear on Write to bit
`0x5A`	PLBERR_ST_PLBM	PLB Error Status	Clear on Write to bit
Hardware Debug			
`0x5B`	SM_ST_PLBM	State Machine States Register	Read Only
`0x5C`	MISC_PLBM	Miscellaneous Control and Status	R/W, Write Only
`0x5D`	CMD_SNIFF_PLBM	Command Sniffer	R/W
`0x5E`	CMD_SNIFFA_PLBM	Command Sniffer Address	R/W
`0x5F`	-	Reserved	-

Registers 0x20 to 0x33: DCRs for the Crossbar

0x20: Interrupt Status Register (IST), Clear on Writes, Read Only

This register contains all the interrupt status bits of the two PLB slave interfaces, PLB Master interface, and the crossbar (see Table 14-10). All register bits are cleared on writes, except those that are marked as read only (RO). Writing a 1 to a clear-on-write bit clears it. The read-only bits are cleared by writing to their corresponding source DCRs. For example, bit 7 is cleared by writing 0s to the PLBS0 FIFO Error Status register.

Note: Even if a particular interrupt is masked, these status bits are still set if the error condition is detected.

Table 14-10: **Bit Definitions for the IST Register**

Bits	Field	Type	Default	Description
0:2	Reserved	-	`000`	Reserved
3	INT_CFG_ERR_S0	RO	0	Configuration or command error, PLBS0. See register `0x39` for further information.
4	INT_MIRQ_S0	RO	0	PLB MIRQ error, PLBS0
5	INT_MRDERR_S0	Clr on Wr	0	Read transaction error, PLBS0. See registers `0x36` through `0x38` for further information.
6	INT_MWRERR_S0	Clr on Wr	0	Write transaction error, PLBS0. See registers `0x36` through `0x38` for further information.
7	INT_FIFO_ERR_S0	RO	0	FIFO error interrupt, PLBS0. See register `0x39` for further information.
8:10	Reserved	-	`000`	Reserved
11	INT_CFG_ERR_S1	RO	0	Configuration or command error, PLBS1. See register `0x49` for further information.
12	INT_MIRQ_S1	RO	0	PLB MIRQ error, PLBS1
13	INT_MRDERR_S1	Clr on Wr	0	Read transaction error, PLBS1. See registers `0x46` through `0x48` for further information.
14	INT_MWRERR_S1	Clr on Wr	0	Write transaction error, PLBS1. See registers `0x46` through `0x48` for further information.
15	INT_FIFO_ERR_S1	RO	0	FIFO error interrupt, PLBS1. See register `0x49` for further information.
16	Reserved	-	0	Reserved
17	INT_CFG_ERR_M	RO	0	Configuration error, PLBM. See register `0x59` for further information.
18	INT_MIRQ_M	RO	0	PLB MIRQ error, PLBM
19	INT_MRDERR_M	Clr on Wr	0	Read transaction error, PLBM. See registers `0x56` through `0x58` for further information.
20	INT_MWRERR_M	Clr on Wr	0	Write transaction error, PLBM. See registers `0x56` through `0x58` for further information.
21	INT_ARB_TOUT_M	Clr on Wr	0	PLB Time-out error, PLBM
22	Reserved	-	0	Reserved

Table 14-10: **Bit Definitions for the IST Register *(Cont'd)***

Bits	Field	Type	Default	Description
23	Reserved	-	0	Reserved
24	INT_FIFO_ERR_M	RO	0	FIFO error interrupt, PLBM. See register `0x59` for further information.
25	INT_FIFO_ERR_XM	RO	0	FIFO error, Crossbar for PLBM. See register `0x58` for further information.
26	INT_FIFO_ERR_MCI	RO	0	FIFO error, Crossbar for MCI. See register `0x5D` for further information.
27:31	Reserved	-	0	Reserved

0x21: Interrupt Mask Register (IMASK), R/W

This register contains the interrupt mask information (see Table 14-11). Clearing a bit to 0 masks the interrupt generation from the corresponding interrupting source in register `0x20`. This register is initialized by embedded processor block attribute INTERCONNECT_IMASK.

Table 14-11: **Bit Definitions for the IMASK Register**

Bits	Field	Default	Description
0:2	Reserved	111	Reserved
3	M_INT_CFG_ERR_S0	1	Interrupt mask for configuration or command error, PLBS0
4	M_INT_MIRQ_S0	1	Interrupt mask for general error, PLBS0
5	M_INT_MRDERR_S0	1	Interrupt mask for read transaction error, PLBS0
6	M_INT_MWRERR_S0	1	Interrupt mask for write transaction error, PLBS0
7	M_INT_FIFO_ERR_S0	1	Interrupt mask for FIFO error, PLBS0
8:10	Reserved	111	Reserved
11	M_INT_CFG_ERR_S1	1	Interrupt mask for configuration or command error, PLBS1
12	M_INT_MIRQ_S1	1	Interrupt mask for general error, PLBS1
13	M_INT_MRDERR_S1	1	Interrupt mask for read transaction error, PLBS1
14	M_INT_MWRERR_S1	1	Interrupt mask for write transaction error, PLBS1
15	M_INT_FIFO_ERR_S1	1	Interrupt mask for FIFO error interrupt, PLBS1
16	Reserved	1	Reserved
17	M_INT_MPLB_ERR_M	1	Interrupt mask for configuration error, PLBM
18	M_INT_MIRQ_M	1	Interrupt mask for general error, PLBM
19	M_INT_MRDERR_M	1	Interrupt mask for read transaction error, PLBM
20	M_INT_MWRERR_M	1	Interrupt mask for write transaction error, PLBM
21	M_INT_ARB_TOUT_M	1	Interrupt mask for PLB time-out error, PLBM
22	Reserved	1	Reserved
23	Reserved	1	Reserved

Table 14-11: **Bit Definitions for the IMASK Register *(Cont'd)***

Bits	Field	Default	Description
24	M_INT_FIFO_ERR_M	1	Interrupt mask for FIFO error interrupt, PLBM
25	M_INT_FIFO_ERR_XM	1	Interrupt mask for FIFO error, Crossbar for PLBM
26	M_INT_FIFO_ERR_MCI	1	Interrupt mask for FIFO error, Crossbar for MCI
27:31	Reserved	5`b1	Reserved

0x23: Crossbar for PLB Master Arbitration Configuration Register (ARB_XBC), R/W

This register configures crossbar arbitration priority and mode operations (see Table 14-12). This register is initialized by embedded processor block attribute PPCM_ARBCONFIG. Arbitration priority values apply to fixed and round-robin arbitration only, with 4 corresponding to the highest priority and 0 to the lowest priority. Values between 5 and 7 are reserved and should not be used due to unpredictable behavior. The five device priority values must be mutually exclusive so that no two or more devices can have the same priority, otherwise there are unpredictable results.

Table 14-12: **Bit Definitions for the ARB_XBC Register**

Bits	Field	Default	Description
0:8	Reserved	0	Reserved
9:11	440ICUR	100	Instruction Read Priority
12	Reserved	0	Reserved
13:15	440DCUW	011	Data Write Priority
16	Reserved	0	Reserved
17:19	440DCUR	010	Data Read Priority
20	Reserved	0	Reserved
21:23	PLBS1	000	PLB Slave 1 Priority
24	Reserved	0	Reserved
25:27	PLBS0	001	PLB Slave 0 Priority
28	Reserved	0	Reserved
29	SYNCTATTR	0	Sync TAttribute (bit 7) enable, if set
30:31	MODE	00	Arbitration Mode. • `00`: For Least Recently Used (LRU) • `01`: For round-robin • `10`: For fixed priority • `11`: Reserved (should not be used, may lead to unpredictable behavior)

0x24: Crossbar for PLB Master FIFO Overflow and Underflow Status Register (FIFOST_XBC), Clear on Writes

This register indicates the FIFO overflow and underflow status for the PLB Master (see Table 14-13). Individual register bits are cleared by writing 1s to them. Bit 31 of the interrupt register is set if any of the FIFO overflow or underflow bit is set. None of these bits should ever be set under normal operating conditions.

Table 14-13: **Bit Definitions for the FIFOST_XBC Register**

Bits	Field	Default	Description
0:27	Reserved	0	Reserved
28	FIFO_OF_RCMDQ	0	Indicates a write command queue overflow, when set
29	FIFO_UF_RCMDQ	0	Indicates a write command queue underflow, when set
30	FIFO_OF_WCMDQ	0	Indicates a read command queue overflow, when set
31	FIFO_UF_WCMDQ	0	Indicates a read command queue underflow, when set

0x26: Crossbar for PLB Master Miscellaneous Control and Status Register (MISC_PLBM), R/W or Write Only

This register contains miscellaneous control and status bits for the PLB Master (see Table 14-14). Read values for write-only bits are always 0s.

Table 14-14: **Bit Definitions for the MISC_PLBM Register**

Bits	Field	Default	Type	Description
0:29	Reserved	0	-	Reserved
30	FIFO_RCMDQ_RST	0	Write Only	Write a 1 to this bit to reset the Read Command Queue
31	FIFO_WCMDQ_RST	0	Write Only	Write a 1 to this bit to reset the Write Command Queue

0x28: Crossbar for MCI Arbitration Configuration Register (ARB_XBM), R/W

This register configures crossbar arbitration priority and mode operations (see Table 14-15). This register is initialized by embedded processor block attribute MI_ARBCONFIG. Arbitration priority values apply to fixed and round-robin arbitration only with 4 corresponding to the highest priority and 0 to the lowest priority. Values between 5 and 7 are reserved and should not be used due to unpredictable behavior. The five device priority values must be mutually exclusive so that no two or more devices can have the same priority, otherwise unpredictable results could occur.

Table 14-15: **Bit Definitions for the ARB_XBM Register**

Bits	Field	Default	Description
0:8	Reserved	0	Reserved
9:11	440ICUR	100	Instruction Read Priority
12	Reserved	0	Reserved
13:15	440DCUW	011	Data Write Priority

Table 14-15: **Bit Definitions for the ARB_XBM Register *(Cont'd)***

Bits	Field	Default	Description
16	Reserved	`0`	Reserved
17:19	440DCUR	`010`	Data Read Priority
20	Reserved	`0`	Reserved
21:23	PLBS1	`000`	PLB Slave 1 Priority
24	Reserved	`0`	Reserved
25:27	PLBS0	`001`	PLB Slave 0 Priority
28:29	Reserved	`0`	Reserved
30:31	MODE	`00`	Arbitration Mode. • `00`: For Least Recently Used (LRU) • `01`: For round-robin • `10`: For fixed priority • `11`: Reserved (should not be used, may lead to unpredictable behavior)

0x29: Crossbar for MCI FIFO Overflow and Underflow Status Register (FIFOST_XBM), Clear on Writes

This register indicates the FIFO overflow and underflow status for the MCI (see Table 14-16). Individual register bits are cleared by writing 1s to them. Bit 31 of the interrupt register is set if the FIFO overflow or underflow bit is set. None of the bits should ever be set under normal operating conditions.

Table 14-16: **Bit Definitions for the FIFOST_XBM Register**

Bits	Field	Default	Description
0:27	Reserved	0	Reserved
28	FIFO_OF_RCMDQ	0	Write command queue overflow, when set
29	FIFO_UF_RCMDQ	0	Write command queue underflow, when set
30	FIFO_OF_WCMDQ	0	Read command queue overflow, when set
31	FIFO_UF_WCMDQ	0	Read command queue underflow, when set

0x2B: Crossbar for MCI Miscellaneous Control and Status Register (MISC_XBM), R/W or Write Only

This register contains miscellaneous control and status bits for the MCI (see Table 14-17). Read values for write-only bits are always 0s.

Table 14-17: **Bit Definitions for the MISC_XBM Register**

Bits	Field	Default	Type	Description
0:29	Reserved	0	-	Reserved
30	FIFO_RCMDQ_RST	0	Write Only	Write a 1 to this bit to reset the Read Command Queue
31	FIFO_WCMDQ_RST	0	Write Only	Write a 1 to this bit to reset the Write Command Queue

0x2D to 0x30: Crossbar Template Registers, R/W

There are four 32-bit template registers for the crossbar (see Table 14-18). Selection of one of the four registers for address mapping is done through the Template Selection Register. Each bit of a 32-bit register corresponds to 128 MByte address space for a total of 4 GB addressing. Traffic is routed to the MCI if the address is within the 128 MB address range that has the template bit set; otherwise the traffic is routed to the PLB Master. These registers are initialized by embedded processor block attributes XBAR_ADDRMAP_TMPL0 through XBAR_ADDRMAP_TMPL3.

Table 14-18: **Crossbar Template Registers**

Address	Mnemonic	Default	Description
`0x2D`	TMPL0_XBAR_MAP	`32'hFFFF_0000`	Template Register 0 for Crossbar
`0x2E`	TMPL1_XBAR_MAP	`32'h0000_0000`	Template Register 1 for Crossbar
`0x2F`	TMPL2_XBAR_MAP	`32'h0000_0000`	Template Register 2 for Crossbar
`0x30`	TMPL3_XBAR_MAP	`32'h0000_0000`	Template Register 3 for Crossbar

0x31: Template Selection Register (TMPL_SEL_REG), R/W

This register is the template selection register for specifying the address mapping template (see Table 14-19). There are 16 x 2-bit entries in this register corresponding to a 16 x 4-GB address space. Each two-bit field identifies which of the four TMPL*_XBAR_MAP registers are used to map crossbar addresses, which of the four TMPL*_PLBS0_MAP registers are used to enable address decoding on SPLB0, and which of the four TMPL*_PLBS1_MAP registers are used to enable address decoding on SPLB1. By default, all of these address template registers are configured so that template 0 controls all crossbar mapping and SPLB interface decoding for the lower 4 GB address space. Because EDK supports only the lower 4 GB space, there is normally no reason for users to use templates 1 through 3. This register is initialized by embedded processor block attribute INTERCONNECT_TMPL_SEL.

Table 14-19: **Bit Definitions for the TMPL_SEL_REG Register**

Bits	Field	Default	Description
0:31	SEL	`32'h3FFF_FFFF`	16 2-bit values for template register selection

DCRs for PLB Slave 0, SPLB 0 (0x34 to 0x43)

0x34: PLB Slave 0 Configuration Register (CFG_PLBS0), R/W

This register configures PLB Slave 0 operation (see Table 14-20). This register is initialized by embedded processor block attribute PPCS0_CONTROL.

Table 14-20: **Bit Definitions for the CFG_PLBS0 Register**

Bits	Field	Default	Description
0	LOCK_SESR	1	When this bit is set, the SESR and SEAR registers (`0x36`, `0x37`, and `0x38`) are locked.
1	Reserved	0	Reserved
2	DMA1_EN	0	• 0: Disable DMA1 • 1: Enable DMA1
3	DMA0_EN	0	• 0: Disable DMA0 • 1: Enable DMA0
4:5	DMA0_PRI	`00`	DMA0 priority • `00`: Lowest priority • `11`: Highest priority
6:7	DMA1_PRI	`00`	DMA1 priority • `00`: Lowest priority • `11`: Highest priority
8	Reserved	0	Reserved
9:11	THRMCI	`011`	Command translation for a read MCI threshold of 8. • `000`: Threshold of 1, a burst is turned into single transfers. • `001`: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • `010`: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • `011`: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • `100`: Threshold of 16, a burst is turned into burst-of-16 transfers, if applicable. • `101` to `111`: Reserved values that can cause unpredictable behaviors.
12	Reserved	0	Reserved
13:15	THRPLBM	`011`	Command translation for a read PLB Master threshold of 8. • `000`: Threshold of 1, a burst is turned into single transfers. • `001`: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • `010`: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • `011`: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • `100`: Threshold of 16, a burst is turned into burst-of-16 transfers, if applicable. • `101` to `111`: Reserved values that can cause unpredictable behaviors.

Table 14-20: **Bit Definitions for the CFG_PLBS0 Register *(Cont'd)***

Bits	Field	Default	Description
16	Reserved	0	Reserved
17:19	THWMCI	011	Command translation for a write MCI threshold of 8. • 000: Threshold of 1, a burst is turned into single transfers. • 001: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • 010: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • 011: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • 100: Threshold of 16, a burst is turned into burst-of-16 transfers, if applicable. • 101 to 111: Reserved values that can cause unpredictable behaviors.
20	Reserved	0	Reserved
21:23	THWPLBM	011	Command translation for a write PLB Master threshold of 8. • 000: Threshold of 1, a burst is turned into single transfers. • 001: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • 010: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • 011: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • 100: Threshold of 16, a burst is turned into burst-of-16 transfers, if applicable. • 101 to 111: Reserved values that can cause unpredictable behaviors.
24	Reserved	0	Reserved
25	LOCKXFER	1	Lock Transfers • 0: Disables lock transfers • 1: Enables lock transfers
26	RPIPE	1	Read Address Pipelining • 0: Disables read address pipelining • 1: Enables read address pipelining
27	WPIPE	0	Write Address Pipelining • 0: Disables write address pipelining • 1: Enables write address pipelining Cleared automatically when bit 28 is 0 to prevent posted write data.
28	WPOST	1	Write Posting • 0: No write posting (early data acknowledge) • 1: Enables write posting Bit 27 is cleared when this bit is 0. Only single transactions are supported when write posting is disabled. The interrupt status flag (bit 3 of Crossbar register 0x20) is set if other types of transactions are received.
29	Reserved	1	Must be set to 1.

Table 14-20: **Bit Definitions for the CFG_PLBS0 Register *(Cont'd)***

Bits	Field	Default	Description
30	AERR_LOG	0	Log ABUS address mismatch error, when set (see bit 2 in register `0x39`)
31	CMD_CHK_DBL	0	Disable command (size) check, when set (see bits 0 and 1 in register `0x39`)

0x36: PLB Slave 0 Error Address Register (SEAR_U_PLBS0), Clear on Writes

This register is cleared by writing to register `0x38`. This register captures the upper 4-bit address of a 36-bit address of a failed transaction (see Table 14-21). The content is valid if bit 0 of register `0x38` is set. A failed transaction corresponds to a command address mismatch or an illegal command. This register is also used by the command sniffer (see registers `0x3D` and `0x3E`).

Table 14-21: **Bit Definitions for the SEAR_U_PLBS0 Register**

Bits	Field	Default	Description
0:27	Reserved	0	Reserved
28:31	U4BIT	0	Upper 4 bits of a 36-bit address

0x37: PLB Slave 0 Error Address Register (SEAR_L_PLBS0), Clear on Writes

This register is cleared by writing to register `0x38`. This register captures the lower 32-bit address of 36-bit address of a failed transaction (see Table 14-22). The content is valid if bit 0 of register `0x38` is set. A failed transaction corresponds to a command address mismatch or an illegal command. This register is also used by the command sniffer (see registers `0x3D` and `0x3E`).

Table 14-22: **Bit Definitions for the SEAR_L_PLBS0 Register**

Bits	Field	Default	Description
0:31	L32BIT	0	Lower 32 bits of a 36-bit address

0x38: PLB Slave 0 Error Status Register (SESR_PLBS0), Clear on Writes

This register captures the transaction qualifiers of a failed transaction (see Table 14-23). A failed transaction corresponds to a command address mismatch or an illegal command. The slave interface only supports the following commands:

- Single transfers
- 4, 8, and 16-word line transfers
- 32-bit, 64-bit, and 128-bit burst transfers

All other commands are considered illegal. Furthermore, if write posting is disabled, only single transfers are supported, any other types of transfers are considered illegal.

This register is also used by the command sniffer (see registers `0x3D` and `0x3E`).

The content is valid when bit 0 is set. See also registers `0x36` and `0x37`. This register is cleared by writing to it. When bit 0 of `0x34` is set, this register is only updated when bit 0 becomes 0. When bit 0 of `0x34` is not set, this register is updated every time an error or sniff event is detected.

Table 14-23: **Bit Definitions for the SESR_PLBS0 Register**

Bits	Field	Default	Description
0	VLD	`1'b0`	Valid • 0: No error detected • 1: Error or sniffed command detected
1	LOCKERR	`1'b0`	M_lockErr from the PLB Master
2:3	Reserved	`2'b0`	Reserved
4:5	MID	`2'b0`	Master ID
6:7	MSIZE	`2'b0`	Master Size
8:10	TYPE	`3'b0`	PLB Type. Only `000` for memory transfers is supported.
11	RNW	`1'b0`	Read/Write. • 0: Write • 1: Read
12:15	SIZE	`4'b0`	PLB Size
16:31	BE	`16'b0`	16-bit Byte Enable

0x39: PLB Slave 0 Miscellaneous Status Register (MISC_ST_PLBS0), Clear on Writes

This register contains miscellaneous status bits for PLB Slave 0 (see Table 14-24). Individual register bits are cleared by writing 1s to those bits that need to be cleared. Bit 3 of the Interrupt Status register is set if the configuration error bit, the illegal command bit, or the address mismatch error bit is set. Bit 7 of the Interrupt Status register is set if any FIFO overflow or underflow bit is set. None of these bits should ever be set under normal operating conditions.

Table 14-24: **Bit Definitions for the MISC_ST_PLBS0 Register**

Bits	Field	Default	Description
0	WPOST_CFG_ERR	0	When this bit is set, a write posting configuration error occurred. No write posting is configured (see register `0x34`) but a line or a burst transfer is detected.
1	ILLEGAL_CMD	0	Illegal command detected. The supported commands include: size = `4'h0`, `4'h1`, `4'h2`, `4'h3`, `4'hA`, `4'hB`, or `4'hC`. Qualified by bit 31 of register `0x34`.
2	ADDR_ERR	0	Address mismatch error. Qualified by bit 30 of register `0x34`.
3:17	Reserved	0	Reserved
18	FIFO_OF_RDAT	0	When set, a Read Data Queue overflow occurred
19	FIFO_UF_RDAT	0	When set, a Read Data Queue underflow occurred
20	FIFO_OF_WDAT	0	When set, a Write Data Queue overflow occurred
21	FIFO_UF_WDAT	0	When set, a Write Data Queue underflow occurred
22	FIFO_OF_SRDQ	0	When set, a Slave Read Queue overflow occurred

Table 14-24: **Bit Definitions for the MISC_ST_PLBS0 Register *(Cont'd)***

Bits	Field	Default	Description
23	FIFO_UF_SRDQ	0	When set, a Slave Read Queue underflow occurred
24	FIFO_OF_SWRQ	0	When set, a Slave Write Queue overflow occurred
25	FIFO_UF_SWRQ	0	When set, a Slave Write Queue underflow occurred
26	FIFO_OF_MRDQ	0	When set, a Master Read Queue overflow occurred
27	FIFO_UF_MRDQ	0	When set, a Master Read Queue underflow occurred
28	FIFO_OF_MWRQ	0	When set, a Master Write Queue overflow occurred
29	FIFO_UF_MWRQ	0	When set, a Master Write Queue underflow occurred
30	FIFO_OF_INCMD	0	When set, an Input Command Queue overflow occurred
31	FIFO_UF_INCMD	0	When set, an Input Command Queue underflow occurred

0x3A: PLB Slave 0 PLB Error Status Register (PLBERR_ST_PLBS0), Clear on Writes

This register contains MIRQ status bits for PLB Slave 0 (see Table 14-25). Individual register bits are cleared by writing 1s to those bits that need to be cleared. Bits 28:31 are PLB MIRQ status bits, which can be set due to either the propagation of the slave MIRQ status or conversion of slave MwrErr into MIRQ because of write posting. Refer to the *PLB Architecture Specification* [Ref 4] for more information on the MIRQ signal.

Table 14-25: **Bit Definitions for the PLBERR_ST_PLBS0 Register**

Bits	Field	Default	Description
0:27	Reserved	0	Reserved
28	PLBS0_M0_MIRQ	0	PLB Slave 0, Master 0 MIRQ
29	PLBS0_M1_MIRQ	0	PLB Slave 0, Master 1 MIRQ
30	PLBS0_M2_MIRQ	0	PLB Slave 0, Master 2 MIRQ
31	PLBS0_M3_MIRQ	0	PLB Slave 0, Master 3 MIRQ

0x3B: PLB Slave 0 State Machine States Register (SM_ST_PLBS0), Read Only

This register indicates the states of the PLB Slave 0 state machine (see Table 14-26). This register is reserved for internal use.

Table 14-26: **Bit Definitions for the SM_ST_PLBS0 Register**

Bits	Field	Default	Description
0:31	Reserved	0	Reserved

0x3C: PLB Slave 0 Miscellaneous Control and Status Register (MISC_PLBS0), R/W, Write Only, or Read Only

This register contains miscellaneous control and status bits for PLB Slave 0 (see Table 14-27). Write-only bits always read as 0s.

Table 14-27: **Bit Definitions for the MISC_PLBS0 Register**

Bits	Field	Default	Type	Description
0	MODE_128N64	1	Read Only	• 0: PLBS0 is in 64-bit mode • 1: PLBS0 is in 128-bit mode
1:24	Reserved	0	-	Reserved
25	FIFO_RDAT_RST	0	Write Only	Write a 1 to this bit to reset the Read Data Queue
26	FIFO_WDAT_RST	0	Write Only	Write a 1 to this bit to reset the Write Data Queue
27	FIFO_SRDQ_RST	0	Write Only	Write a 1 to this bit to reset the Slave Read Queue
28	FIFO_SWRQ_RST	0	Write Only	Write a 1 to this bit to reset the Slave Write Queue
29	FIFO_MRDQ_RST	0	Write Only	Write a 1 to this bit to reset the Master Read Queue
30	FIFO_MWRQ_RST	0	Write Only	Write a 1 to this bit to reset the Master Write Queue
31	FIFO_INCMD_RST	0	Write Only	Write a 1 to this bit to reset the Input Command Queue

0x3D: PLB Slave 0 Command Sniffer Register (CMD_SNIFF_PLBS0), R/W

This register contains the description of a command (the address is specified in `0x3E`) that is to be monitored (see Table 14-28). The result is placed in registers `0x38` through `0x3A`. This register is used for debugging purposes.

Table 14-28: **Bit Definitions for the CMD_SNIFF_PLBS0 Register**

Bits	Field	Default	Description
0	ENABLE	0	• 0: Command capture is disabled • 1: Command capture is enabled
1:3	Reserved	000	Reserved
4:7	SIZE	0000	PLB size value to be matched
8	RNW	0	PLB RNW to be matched
9:11	MID	000	Master ID (0 – 4) to be matched
12	SPLBNDMA	0	• 0: Command from DMA engine to be matched • 1: Command from SPLB to be matched
13	Reserved	0	Reserved
14:15	SPLB_MID	00	FPGA logic master's ID (0 – 3) to be matched
16:17	SSIZE	00	SSIZE to be matched
18:24	Reserved	7'b0	Reserved
25	SIZE_EN	0	• 0: Disable size match • 1: Enable size match

Table 14-28: **Bit Definitions for the CMD_SNIFF_PLBS0 Register *(Cont'd)***

Bits	Field	Default	Description
26	RNW_EN	0	• 0: Disable RNW match • 1: Enable RNW match
27	MID_EN	0	• 0: Disable master ID match • 1: Enable master ID match
28	Reserved	0	Reserved
29	Reserved	0	Reserved
30	SSIZE_EN	0	• 0: Disable ssize match • 1: Enable ssize match
31	ADDR_EN	0	• 0: Disable address match • 1: Enable address match

0x3E: PLB Slave 0 Command Sniffer Address Register (CMD_SNIFFA_PLBS0), R/W

This register, used in conjunction with register `0x3D`, contains the address for command sniffing (see Table 14-29).

Table 14-29: **Bit Definitions for the CMD_SNIFFA_PLBS0 Register**

Bits	Field	Default	Description
0:31	L32BIT	0	Lower 32 bits of a 36-bit address

0x40 to 0x43: PLB Slave 0 Template Registers, R/W

Table 14-30 lists the set of four 32-bit template registers for PLB Slave 0. Selection of one of four registers for address mapping is done through the Template Selection Register. Each bit of a 32-bit register corresponds to a 128 MB address space for a 4 GB addressing. Set a bit to 1 to enable the corresponding 128 MB address space. These registers are initialized by embedded processor block attributes PPCS0_ADDRMAP_TMPL0 through PPCS0_ADDRMAP_TMPL3.

Table 14-30: **PLB Slave 0 Template Registers**

Address	Mnemonic	Default	Description
`0x40`	TMPL0_PLBS0_MAP	`32'hFFFF_FFFF`	Template Register 0 for PLB Slave 0
`0x41`	TMPL1_PLBS0_MAP	`32'hFFFF_FFFF`	Template Register 1 for PLB Slave 0
`0x42`	TMPL2_PLBS0_MAP	`32'hFFFF_FFFF`	Template Register 2 for PLB Slave 0
`0x43`	TMPL3_PLBS0_MAP	`32'hFFFF_FFFF`	Template Register 3 for PLB Slave 0

DCRs for PLB Slave 1, SPLB 1 (0x44 to 0x53)

0x44: PLB Slave 1 Configuration Register (CFG_PLBS1), R/W

This register configures PLB Slave 1 operation (see Table 14-31). This register is initialized by embedded processor block attribute PPCS1_CONTROL.

Table 14-31: **Bit Definitions for the CFG_PLBS1 Registers**

Bits	Field	Default	Description
0	LOCK_SESR	`1`	When this bit is set, the SESR and SEAR registers (`0x46`, `0x47`, and `0x48`) are locked.
1	Reserved	`0`	Reserved
2	DMA3_EN	`0`	• 0: Disable DMA3 • 1: Enable DMA3
3	DMA2_EN	`0`	• 0: Disable DMA2 • 1: Enable DMA2
4:5	DMA2_PRI	`00`	DMA2 priority • `00`: Lowest priority • `11`: Highest priority
6:7	DMA3_PRI	`00`	DMA3 priority • `00`: Lowest priority • `11`: Highest priority
8	Reserved	`0`	Reserved
9:11	THRMCI	`011`	Command translation for a read MCI threshold of 8. • `000`: Threshold of 1, a burst is turned into single transfers. • `001`: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • `010`: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • `011`: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • `100`: Threshold of 16, a burst of turned into burst-of-16 transfers, if applicable. • `101` to `111` are reserved values and can cause unpredictable behaviors.
12	Reserved	`0`	Reserved
13:15	THRPLBM	`011`	Command translation for a read PLB Master threshold of 8. • `000`: Threshold of 1, a burst is turned into single transfers. • `001`: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • `010`: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • `011`: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • `100`: Threshold of 16, a burst of turned into burst-of-16 transfers, if applicable. • `101` to `111` are reserved values and can cause unpredictable behaviors.

Table 14-31: **Bit Definitions for the CFG_PLBS1 Registers *(Cont'd)***

Bits	Field	Default	Description
16	Reserved	0	Reserved
17:19	THWMCI	011	Command translation for a write MCI threshold of 8. • 000: Threshold of 1, a burst is turned into single transfers. • 001: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • 010: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • 011: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • 100: Threshold of 16, a burst of turned into burst-of-16 transfers, if applicable. • 101 to 111 are reserved values and can cause unpredictable behaviors.
20	Reserved	0	Reserved
21:23	THWPLBM	011	Command translation for a write PLB Master threshold of 8. • 000: Threshold of 1, a burst is turned into single transfers. • 001: Threshold of 2, a burst is turned into burst-of-2 transfers, if applicable. • 010: Threshold of 4, a burst is turned into burst-of-4 transfers, if applicable. • 011: Threshold of 8, a burst is turned into burst-of-8 transfers, if applicable. • 100: Threshold of 16, a burst of turned into burst-of-16 transfers, if applicable. • 101 to 111 are reserved values and can cause unpredictable behaviors.
24	Reserved	0	Reserved
25	LOCKXFER	1	Lock Transfer. • 0: Disables lock transfers • 1: Enables lock transfers
26	RPIPE	1	Read Address Pipelining. • 0: Disables read address pipelining • 1: Enables read address pipelining
27	WPIPE	0	Write Address Pipelining. • 0: Disables write address pipelining • 1: Enables write address pipelining Cleared automatically if bit 28 is 0 to prevent posted write data.
28	WPOST	1	Write Posting. • 0: No write posting (early data ack) • 1: Enable write posting Bit 27 is cleared if this bit is 0. Only single transactions are supported if write posting is disabled. Interrupt status flag (bit 11 of Crossbar register 0x20) is set if other types of transactions are received.
29	Reserved	1	Must be set to 1.

Table 14-31: **Bit Definitions for the CFG_PLBS1 Registers *(Cont'd)***

Bits	Field	Default	Description
30	AERR_LOG	0	Log ABUS address mismatch error, if set (see bit 2, register `0x49`)
31	CMD_CHK_DBL	0	Disable command (size) check, if set (see bits 0 and 1, register `0x49`)

0x46: PLB Slave 1 Error Address Register (SEAR_U_PLBS1), Clear on Writes

This register is cleared by writing to register `0x48`. This register captures the upper 4-bit address of a 36-bit address of a failed transaction (see Table 14-32). The content is valid if bit 0 of register `0x48` is set. A failed transaction corresponds to a command address mismatch or an illegal command. This register is also used by the command sniffer (see registers `0x4D` and `0x4E`).

Table 14-32: **Bit Definitions for the SEAR_U_PLBS1 Register**

Bits	Field	Default	Description
0:27	Reserved	0	Reserved
28:31	U4BIT	0	Upper 4 bits of a 36-bit address

0x47: PLB Slave 1 Error Address Register (SEAR_L_PLBS1), Clear on Writes

This register is cleared by writing to register `0x48`. This register captures the lower 32-bit address of the 36-bit address of a failed transaction (see Table 14-33). This content is valid if bit 0 of register `0x48` is set. A failed transaction corresponds to a command address mismatch or an illegal command. This register is also used by the command sniffer (see registers `0x4D` and `0x4E`).

Table 14-33: **Bit Definitions for the SEAR_L_PLBS1 Register**

Bits	Field	Default	Description
0:31	L32BIT	0	Lower 32 bits of a 36-bit address

0x48: PLB Slave 1 Error Status Register (SESR_PLBS1), Clear on Writes

This register captures the transaction qualifiers of a failed transaction (see Table 14-34). A failed transaction corresponds to a command address mismatch or an illegal command. The slave interface only supports the following commands:

- Single transfers
- 4, 8, and 16-word line transfers
- 32-bit, 64-bit, and 128-bit burst transfers

All other commands are considered illegal. Furthermore, if write posting is disabled, only single transfers are supported, and any other types of transfers are considered illegal.

This register is also used by the command sniffer (see registers `0x4D` and `0x4E`).

The content is valid if bit 0 is set. See also registers `0x46` and `0x47`. This register is cleared by writing to it. If bit 0 of register `0x44` is set, this register is only updated when bit 0 becomes 0. If bit 0 of register `0x44` is not set, this register is updated every time an error or sniff event is detected.

Table 14-34: **Bit Definitions for the SESR_PLBS1 Register**

Bits	Field	Default	Description
0	VLD	`1'b0`	Valid • 0: No error detected • 1: Error or sniffed command detected
1	LOCKERR	`1'b0`	M_lockErr from the PLB Master
2:3	Reserved	`2'b0`	Reserved
4:5	MID	`2'b0`	Master ID
6:7	MSIZE	`2'b0`	Master Size
8:10	TYPE	`3'b0`	PLB Type. Only `000` for memory transfers is supported.
11	RNW	`1'b0`	Read/Write. • 0: Write • 1: Read
12:15	SIZE	`4'b0`	PLB Size
16:31	BE	`16'b0`	16-bit Byte Enable

0x49: PLB Slave 1 Miscellaneous Status Register (MISC_ST_PLBS1), Clear on Writes

This register contains miscellaneous status bits for PLB Slave 1 (see Table 14-35). Individual register bits are cleared by writing 1s to those bits that need to be cleared. Bit 11 of the Interrupt Status register is set if the configuration error bit, the illegal command bit, or the address mismatch error bit is set. Bit 15 of the Interrupt Status register is set if any FIFO overflow or underflow bit is set. None of these bits should ever be set under normal operating conditions.

Table 14-35: **Bit Definitions for the MISC_ST_PLBS1 Register**

Bits	Field	Default	Description
0	WPOST_CFG_ERR	0	When this bit is set, a write posting configuration error occurred. No write posting is configured (see register `0x44`) but a line or a burst transfer is detected.
1	ILLEGAL_CMD	0	Illegal command detected. The supported commands include: size = `4'h0`, `4'h1`, `4'h2`, `4'h3`, `4'hA`, `4'hB`, or `4'hC`. Qualified by bit 31 of register `0x44`.
2	ADDR_ERR	0	Address mismatch error. Qualified by bit 30 of register `0x44`.
3:17	Reserved	0	Reserved
18	FIFO_OF_RDAT	0	When this bit is set, a Read Data Queue overflow occurred
19	FIFO_UF_RDAT	0	When this bit is set, a Read Data Queue underflow occurred

Table 14-35: **Bit Definitions for the MISC_ST_PLBS1 Register *(Cont'd)***

Bits	Field	Default	Description
20	FIFO_OF_WDAT	0	When this bit is set, a Write Data Queue overflow occurred
21	FIFO_UF_WDAT	0	When this bit is set, a Write Data Queue underflow occurred
22	FIFO_OF_SRDQ	0	When this bit is set, a Slave Read Queue overflow occurred
23	FIFO_UF_SRDQ	0	When this bit is set, a Slave Read Queue underflow occurred
24	FIFO_OF_SWRQ	0	When this bit is set, a Slave Write Queue overflow occurred
25	FIFO_UF_SWRQ	0	When this bit is set, a Slave Write Queue underflow occurred
26	FIFO_OF_MRDQ	0	When this bit is set, a Master Read Queue overflow occurred
27	FIFO_UF_MRDQ	0	When this bit is set, a Master Read Queue underflow occurred
28	FIFO_OF_MWRQ	0	When this bit is set, a Master Write Queue overflow occurred
29	FIFO_UF_MWRQ	0	When this bit is set, a Master Write Queue underflow occurred
30	FIFO_OF_INCMD	0	When this bit is set, an Input Command Queue overflow occurred
31	FIFO_UF_INCMD	0	When this bit is set, an Input Command Queue underflow occurred

0x4A: PLB Slave 1 PLB Error Status Register (PLBERR_ST_PLBS1), Clear on Writes

This register contains the MIRQ status bits for PLB Slave 1 (see Table 14-36). Individual register bits are cleared by writing 1s to those bits that need to be cleared. Bits 28:31 are PLB MIRQ status bits, which can be set due to either the propagation of the slave MIRQ status or conversion of slave MwrErr into MIRQ because of write posting.

Table 14-36: **Bit Definitions for the PLBERR_ST_PLBS1 Register**

Bits	Field	Default	Description
0:27	Reserved	0	Reserved
28	PLBS1_M0_MIRQ	0	PLB Slave 1, Master 0 MIRQ
29	PLBS1_M1_MIRQ	0	PLB Slave 1, Master 1 MIRQ
30	PLBS1_M2_MIRQ	0	PLB Slave 1, Master 2 MIRQ
31	PLBS1_M3_MIRQ	0	PLB Slave 1, Master 3 MIRQ

0x4B: PLB Slave 1 State Machine States Register (SM_ST_PLBS1), Read Only

This register indicates the states of the state machine for PLB Slave 1 (see Table 14-37). This register is reserved for internal use.

Table 14-37: **Bit Definitions for the SM_ST_PLBS1 Register**

Bits	Field	Default	Description
0:31	Reserved	0	Reserved

0x4C: PLB Slave 1 Miscellaneous Control and Status Register (MISC_PLBS1), R/W, Write Only, or Read Only

This register contains miscellaneous control and status bits for PLB Slave 1 (see Table 14-38). Write-only bits always read as 0s.

Table 14-38: **Bit Definitions for the MISC_PLBS1 Register**

Bits	Field	Default	Type	Description
0	MODE_128N64	1	Read Only	• 0: PLBS1 is in 64-bit mode • 1: PLBS1 is in 128-bit mode
1:24	Reserved	0	-	Reserved
25	FIFO_RDAT_RST	0	Write Only	Write a 1 to this bit to reset the Read Data Queue
26	FIFO_WDAT_RST	0	Write Only	Write a 1 to this bit to reset the Write Data Queue
27	FIFO_SRDQ_RST	0	Write Only	Write a 1 to this bit to reset the Slave Read Queue
28	FIFO_SWRQ_RST	0	Write Only	Write a 1 to this bit to reset the Slave Write Queue
29	FIFO_MRDQ_RST	0	Write Only	Write a 1 to this bit to reset the Master Read Queue
30	FIFO_MWRQ_RST	0	Write Only	Write a 1 to this bit to reset the Master Write Queue
31	FIFO_INCMD_RST	0	Write Only	Write a 1 to this bit to reset the Input Command Queue

0x4D: PLB Slave 1 Command Sniffer Register (CMD_SNIFF_PLBS1), R/W

This register contains the description of a command (whose address is specified in register `0x4E`) that is to be monitored (see Table 14-39). The results are placed in registers `0x48` through `0x4A`. This register is used for debugging purposes.

Table 14-39: **Bit Definitions for the CMD_SNIFF_PLBS1 Register**

Bits	Field	Default	Description
0	ENABLE	0	• 0: Command capture disabled • 1: Command command enabled
1:3	Reserved	000	Reserved
4:7	SIZE	0000	PLB size value to be matched

Table 14-39: **Bit Definitions for the CMD_SNIFF_PLBS1 Register *(Cont'd)***

Bits	Field	Default	Description
8	RNW	0	PLB RNW to be matched
9:11	MID	000	Master ID (0 – 4) to be matched
12	SPLBNDMA	0	• 0: Command from DMA engine to be matched • 1: Command from SPLB to be matched
13	Reserved	0	Reserved
14:15	SPLB_MID	00	FPGA logic master's ID (0 – 3) to be matched
16:17	SSIZE	00	SSIZE to be matched
18:24	Reserved	7'h0	Reserved
25	SIZE_EN	0	• 0: Disable size match • 1: Enable size match
26	RNW_EN	0	• 0: Disable RNW match • 1: Enable RNW match
27	MID_EN	0	Enable master ID match, if set • 0: Disable master ID match • 1: Enable master ID match
28	Reserved	0	Reserved
29	Reserved	0	Reserved
30	SSIZE_EN	0	• 0: Disable ssize match • 1: Enable ssize match
31	ADDR_EN	0	• 0: Disable address match • 1: Enable address match

0x4E: PLB Slave 1 Command Sniffer Address (CMD_SNIFFA_PLBS1), R/W

This register, used in conjunction with register `0x4D`, contains the address (lower 32 bits) for command sniffing (see Table 14-40).

Table 14-40: **Bit Definitions for the CMD_SNIFFA_PLBS1 Register**

Bits	Field	Default	Description
0:31	L32BIT	0	Lower 32-bit of a 36-bit address

0x50 to 0x53: PLB Slave 1 Template Registers, R/W

Table 14-41 lists the set of four 32-bit template registers for PLB Slave 1. Selection of one of four registers for address mapping is done through the Template Selection Register. Each bit of a 32-bit register corresponds to 128 MB address space for a total of 4 GB addressing. Set a bit to 1 to enable the corresponding 128 MB address space. These registers are initialized by embedded processor block attributes PPCS1_ADDRMAP_TMPL0 through PPCS1_ADDRMAP_TMPL3.

Table 14-41: **PLB Slave 1 Template Registers**

Address	Mnemonic	Default	Description
`0x50`	TMPL0_PLBS1_MAP	`32'hFFFF_FFFF`	Template Register 0 for PLB Slave 1
`0x51`	TMPL1_PLBS1_MAP	`32'hFFFF_FFFF`	Template Register 1 for PLB Slave 1
`0x52`	TMPL2_PLBS1_MAP	`32'hFFFF_FFFF`	Template Register 2 for PLB Slave 1
`0x53`	TMPL3_PLBS1_MAP	`32'hFFFF_FFFF`	Template Register 3 for PLB Slave 1

DCRs for PLB Master, MPLB (0x54 to 0x5F)

0x54: PLB Master Configuration Register (CFG_PLBM), R/W

This register configures PLB Master operation (see Table 14-42). This register is initialized by embedded processor block attribute PPCM_CONTROL.

Table 14-42: **Bit Definitions for the CFG_PLBM Register**

Bits	Field	Default	Description
0	LOCK_SESR	1	When this bit is set, the SESR and SEAR registers (`0x56`, `0x57`, and `0x58`) are locked.
1:22	Reserved	0	Reserved
23	Reserved	0	Must be set to 0.
24	XBAR_PRIORITY_ENA	1	• 0: Priority is disabled during crossbar arbitration • 1: Priority is enabled during crossbar arbitration
25	Reserved	0	Reserved (can lead to unexpected behavior, if set to 1).
26	SL_ETERM_MODE	0	When this bit is set, slave early burst termination is supported. Bits 28 and 29 are cleared automatically when this bit is set. This mode prevents R/W command re-ordering.
27	LOCKXFER	1	Lock Transfers. • 0: Disables lock transfers • 1: Enables lock transfers
28	RPIPE	1	Read Address Pipelining. • 0: Disables read address pipelining • 1: Enables read address pipelining Can be cleared directly or through bit 26.
29	WPIPE	1	Write Address Pipelining. • 0: Disables write address pipelining • 1: Enables write address pipelining Can be cleared directly or through bit 26. This bit is cleared if bit 30 is 0 to prevent posted write data.

Table 14-42: **Bit Definitions for the CFG_PLBM Register *(Cont'd)***

Bits	Field	Default	Description
30	WPOST	1	Write Posting. • 0: No write posting (early data ack) • 1: Enable write posting Bit 29 is cleared if this bit is 0. Only single transactions are supported if write posting is disabled. Interrupt status flag (bit 17 of Crossbar register `0x20`) is set if other types of transactions are received.
31	Reserved	1	Must be set to 1.

0x56: FPGA Logic Slave Error Address Register (FSEAR_U_PLBM), Clear on Writes

This register is cleared by writing to register `0x58`. This register captures the upper 4-bit address of a 36-bit address of a failed transaction (see Table 14-43). The content is valid if bit 0 of register `0x58` is set. A failed transaction corresponds to one of the following conditions: address time-out, data time-out, PLB slave MwrErr, or PLB slave MrdErr. This register is also used by the command sniffer (see registers `0x5D` – `0x5E`).

Table 14-43: **Bit Definitions for the FSEAR_U_PLBM Register**

Bits	Field	Default	Description
0:27	Reserved	0	Reserved
28:31	U4BIT	0	Upper 4 bits of a 36-bit address

0x57: FPGA Logic Slave Error Address Register (FSEAR_L_PLBM), Clear on Writes

This register is cleared by writing to register `0x58`. This register captures the lower 32-bit address of a 36-bit address of a failed transaction (see Table 14-44). The content is valid if bit 0 of register `0x58` is set. A failed transaction corresponds to one of the following conditions: address time-out, data time-out, PLB slave MwrErr, or PLB slave MrdErr. This register is also used by the command sniffer (see registers `0x5D` – `0x5E`).

Table 14-44: **Bit Definitions for the FSEAR_L_PLBM Register**

Bits	Field	Default	Description
0:31	L32BIT	0	Lower 32 bits of a 36-bit address

0x58: FPGA Logic Slave Error Status Register (FSESR_PLBM), Clear on Writes

This register is cleared by writing to it. This register captures the transaction qualifiers of a failed transaction (see Table 14-45). A failed transaction corresponds to one of the following conditions: address time-out, data time-out, PLB slave MwrErr, or PLB slave MrdErr.

This register is also used by the command sniffer (see registers `0x5D` – `0x5E`).

The content is valid if bit 0 is set. See also registers `0x56` and `0x57`. If bit 0 of register `0x54` is set, this register is only updated when bit 0 becomes 0. If bit 0 of register `0x54` is not set, the register is updated every time an error or sniff event is detected.

Table 14-45: **Bit Definitions for the FSESR_PLBM Register**

Bits	Field	Default	Description
0	VLD	0	Valid • 0: No error detected • 1: Error or sniffed command detected
1	LOCKERR	0	M_lockErr to the PLB slave
2	PLBS_DMA	0	• 1: Command from PLB slave 0 or 1 • 0: Command from a DMA engine. Value is only valid if MID is 3 or 4.
3:5	MID	`3'b0`	Master ID. • `000`: ICUR • `001`: DCUW • `010`: DCUR • `011`: PLBS0 • `100`: PLBS1
6:7	SSIZE	`2'b0`	Slave size (`00`, `01`, or `10`). `11` indicates address time-out.
8:10	TYPE	`3'b0`	PLB Type. Only `000` for memory transfers is supported.
11	RNW	0	Read/Write. • 0: Write • 1: Read
12:15	SIZE	`4'b0`	PLB Size
16:31	BE	`16'b0`	16-bit Byte Enable

0x59: PLB Master Miscellaneous Status Register (MISC_ST_PLBM), Clear on Writes

This register contains miscellaneous status bits for the PLB Master (see Table 14-46). Individual register bits are cleared by writing 1s to those bits that need to be cleared. Bit 17 of the Interrupt Status register is set if either or both of the configuration error bits are set. Bit 24 of the Interrupt Status register is set if any FIFO overflow or underflow bit is set. None of these bits should ever be set under normal operating conditions.

Table 14-46: **Bit Definitions for the MISC_ST_PLBM Register**

Bits	Field	Default	Description
0	WPOST_CFG_ERR	0	When this bit is set, a write posting configuration error occurred. No write posting is configured (see register `0x54`) but a line or a burst transfer is detected.
1	ETERM_CFG_ERR	0	When this bit is set, a slave early burst termination configuration error occurred. Early termination is not configured (see register `0x54`), but early termination is detected.
2:17	Reserved	0	Reserved
18	FIFO_OF_RDAT	0	When set, a Read Data Queue overflow occurred
19	FIFO_UF_RDAT	0	When set, a Read Data Queue underflow occurred
20	FIFO_OF_WDAT	0	When set, a Write Data Queue overflow occurred

Table 14-46: **Bit Definitions for the MISC_ST_PLBM Register *(Cont'd)***

Bits	Field	Default	Description
21	FIFO_UF_WDAT	0	When set, a Write Data Queue underflow occurred
22	FIFO_OF_SRDQ	0	When set, a Slave Read Queue overflow occurred
23	FIFO_UF_SRDQ	0	When set, a Slave Read Queue underflow occurred
24	FIFO_OF_SWRQ	0	When set, a Slave Write Queue overflow occurred
25	FIFO_UF_SWRQ	0	When set, a Slave Write Queue underflow occurred
26	FIFO_OF_MRDQ	0	When set, a Master Read Queue overflow occurred
27	FIFO_UF_MRDQ	0	When set, a Master Read Queue underflow occurred
28	FIFO_OF_MWRQ	0	When set, a Master Write Queue overflow occurred
29	FIFO_UF_MWRQ	0	When set, a Master Write Queue underflow occurred
30	FIFO_OF_INCMD	0	When set, an Input Command Queue overflow occurred
31	FIFO_UF_INCMD	0	When set, an Input Command Queue underflow occurred

0x5A: PLB Master PLB Error Status Register (PLBERR_ST_PLBM), Read Only or Clear on Writes

This register contains MIRQ status bits for the PLB Master (see Table 14-47). Bits 19:31 are PLB MIRQ status bits that can be set due to either the propagation of the slave MIRQ status or conversion of slave MwrErr into MIRQ because of write posting.

- If the slave PLB MIRQ signal, which is latched at the slave, is set, all MIRQ bits in the register are set. They are read only, so they are cleared when the PLB MIRQ is cleared.
- If the slave PLB MwrErr, which is a pulse, is set and write posting is enabled, one of the MIRQ bits is set. In this case, the register bit is cleared by writing a 1 to the bit.

Bits 19 and 20 correspond to MIRQ errors for writes that originated from the DMA engines in SPLB0/1.

Table 14-47: **Bit Definitions for the PLBERR_ST_PLBM Register**

Bits	Field	Default	Description
0:18	Reserved	0	Reserved
19	PLBS0_DMA_MIRQ	0	PLB Slave 0, DMA MIRQ
20	PLBS1_DMA_MIRQ	0	PLB Slave 1, DMA MIRQ
21	C440_MIRQ_ICUR	0	Processor ICUR MIRQ
22	C440_MIRQ_DCUW	0	Processor DCUW MIRQ
23	C440_MIRQ_DCUR	0	Processor DCUR MIRQ
24	PLBS0_M0_MIRQ	0	PLB Slave 0, Master 0 MIRQ
25	PLBS0_M1_MIRQ	0	PLB Slave 0, Master 1 MIRQ
26	PLBS0_M2_MIRQ	0	PLB Slave 0, Master 2 MIRQ
27	PLBS0_M3_MIRQ	0	PLB Slave 0, Master 3 MIRQ
28	PLBS1_M0_MIRQ	0	PLB Slave 1, Master 0 MIRQ

Table 14-47: **Bit Definitions for the PLBERR_ST_PLBM Register *(Cont'd)***

Bits	Field	Default	Description
29	PLBS1_M1_MIRQ	0	PLB Slave 1, Master 1 MIRQ
30	PLBS1_M2_MIRQ	0	PLB Slave 1, Master 2 MIRQ
31	PLBS1_M3_MIRQ	0	PLB Slave 1, Master 3 MIRQ

0x5B: PLB Master State Machine States Register (SM_ST_PLBM), Read Only

This register indicates the states of the state machine for the PLB Master (see Table 14-48). This register is reserved.

Table 14-48: **Bit Definitions for the SM_ST_PLBM Register**

Bits	Field	Default	Description
0:31	Reserved	0	Reserved

0x5C: PLB Master Miscellaneous Control and Status Register (MISC_PLBM), R/W or Write Only

This register contains miscellaneous control and status bits for the PLB Master (see Table 14-49). Write-only bits always read as 0s.

Table 14-49: **Bit Definitions for the MISC_PLBM Register**

Bits	Field	Default	Type	Description
0	Reserved	0	Write Only	Reserved
1:2	FLUSH_MODE	`00`	R/W	Flush mode select • `00`: Automatic addrAck time-out flush • `01` - `10`: Reserved • `11`: No flush
3	Reserved	0	R/W	Reserved
4:24	Reserved	0	-	Reserved
25	FIFO_RDAT_RST	0	Write Only	Write a 1 to this bit to reset the Read Data Queue
26	FIFO_WDAT_RST	0	Write Only	Write a 1 to this bit to reset the Write Data Queue
27	FIFO_SRDQ_RST	0	Write Only	Write a 1 to this bit to reset the Slave Read Queue
28	FIFO_SWRQ_RST	0	Write Only	Write a 1 to this bit to reset the Slave Write Queue
29	FIFO_MRDQ_RST	0	Write Only	Write a 1 to this bit to reset the Master Read Queue
30	FIFO_MWRQ_RST	0	Write Only	Write a 1 to this bit to reset the Master Write Queue
31	FIFO_INCMD_RST	0	Write Only	Write a 1 to this bit to reset the Input Command Queue

0x5D: PLB Master Command Sniffer Register (CMD_SNIFF_PLBM), R/W

This register contains the description of a command (whose address is specified in register `0x5E`) that is to be monitored (see Table 14-50). The results are placed in registers `0x58` through `0x5A`. This register is used for debugging purposes.

Table 14-50: **Bit Definitions for the CMD_SNIFF_PLBM Register**

Bits	Field	Default	Description
0	ENABLE	0	• 0: Command capture is disabled • 1: Command capture is enabled
1:3	Reserved	`000`	Reserved
4:7	SIZE	`0000`	PLB size value to be matched
8	RNW	`0`	PLB RNW to be matched
9:11	MID	`000`	Master ID (0 – 4) to be matched
12	SPLBNDMA	0	• 0: Command from DMA engine to be matched • 1: Command from SPLB to be matched
13	Reserved	`0`	Reserved
14:15	SPLB_MID	`00`	FPGA logic master's ID (0 – 3) to be matched
16:17	SSIZE	`00`	SSIZE to be matched
18:24	Reserved	`7'h0`	Reserved
25	SIZE_EN	0	• 0: Disable size match • 1: Enable size match
26	RNW_EN	0	• 0: Disable RNW match • 1: Enable RNW match
27	MID_EN	0	• 0: Disable master ID match • 1: Enable master ID match
28	SPLBNDMA_EN	0	• 0: Disable SPLBndma match • 1: Enable SPLBndma match
29	SPLB_MID_EN	0	• 0: Disable SPLB_MID match • 1: Enable SPLB_MID match
30	SSIZE_EN	0	• 0: Disable ssize match • 1: Enable ssize match
31	ADDR_EN	0	• 0: Disable address match • 1: Enable address match

0x5E: PLB Master Command Sniffer Address (CMD_SNIFFA_PLBM), R/W

This register, used in conjunction with register `0x5D`, contains the ABUS address (lower 32 bits) for command sniffing (see Table 14-51).

Table 14-51: **Bit Definitions for the CMD_SNIFFA_PLBM Register**

Bits	Field	Default	Description
0:31	L32BIT	0	Lower 32 bits of a 36-bit address

DMA Engines (0x80 – 0xDF)

Table 14-52 lists the address map for the DCRs.

Table 14-52: **DCR Address Map**

DCR Addresses	Mnemonic	Register Description	Direction
`0x80,0x98,0xB0,0xC8`	TX_NXTDESC_PTR	TX Next Descriptor Pointer	RW(1)
`0x81,0x99,0xB1,0xC9`	TX_CURBUF_ADDR	TX Current Buffer Address Register	RW(1)
`0x82,0x9A,0xB2,0xCA`	TX_CURBUF_LENGTH	TX Current Buffer Length Register	RW(1)
`0x83,0x9B,0xB3,0xCB`	TX_CURDESC_PTR	TX Current Descriptor Pointer	RW
`0x84,0x9C,0xB4,0xCC`	TX_TAILDESC_PTR	TX Tail Descriptor Pointer	RW
`0x85,0x9D,0xB5,0xCD`	TX_CHANNEL_CTRL	TX Channel Control Register	RW
`0x86,0x9E,0xB6,0xCE`	TX_IRQ_REG	TX Interrupt Register	RD-ACK
`0x87,0x9F,0xB7,0xCF`	TX_STATUS_REG	TX Status Register	RW(1)
`0x88,0xA0,0xB8,0xD0`	RX_NXTDESC_PTR	RX Next Descriptor Pointer	RW(1)
`0x89,0xA1,0xB9,0xD1`	RX_CURBUF_ADDR	RX Current Buffer Address Register	RW(1)
`0x8A,0xA2,0xBA,0xD2`	RX_CURBUF_LENGTH	RX Current Buffer Length Register	RW(1)
`0x8B,0xA3,0xBB,0xD3`	RX_CURDESC_PTR	RX Current Descriptor Pointer	RW
`0x8C,0xA4,0xBC,0xD4`	RX_TAILDESC_PTR	RX Tail Descriptor Pointer	RW
`0x8D,0xA5,0xBD,0xD5`	RX_CHANNEL_CTRL	RX Channel Control Register	RW
`0x8E,0xA6,0xBE,0xD6`	RX_IRQ_REG	RX Interrupt Register	RD-ACK
`0x8F,0xA7,0xBF,0xD7`	RX_STATUS_REG	RX Status Register	RW(1)
`0x90,0xA8,0xC0,0xD8`	DMA_CONTROL_REG	DMA Control Register	RW

Notes:

1. These registers are loaded from the descriptors and updated dynamically by the DMA engine. As such, they should not be written during normal operation. Writing them is made available for debug purposes only.
2. See also the DMA enable and DMA priority fields of the SPLB0 and SPLB1 configuration registers (CFG_PLBS0/1) Table 4-6, page 108 in Chapter 4.

Register 0x80, 0x98, 0xB0, 0xC8 – TX Next Descriptor Pointer (TX_NXTDESC_PTR)

Figure 14-5 shows the TX Next Descriptor Pointer. Table 14-53 defines the bits in this pointer.

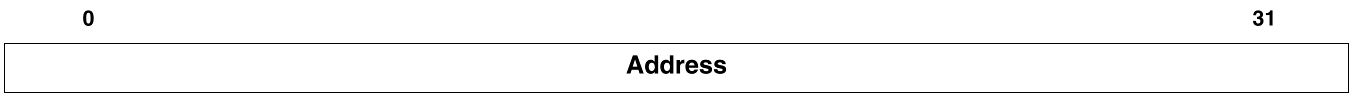


Figure 14-5: **TX Next Descriptor Pointer**

Table 14-53: **Bit Description for TX Next Descriptor Pointer**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	Contains the address of the next descriptor to be fetched. Must be eight-word aligned.

Register 0x81, 0x99, 0xB1, 0xC9 – TX Current Buffer Address (TX_CURBUF_ADDR)

Figure 14-6 shows the TX Current Buffer Address register. Table 14-54 defines the bits in this register.

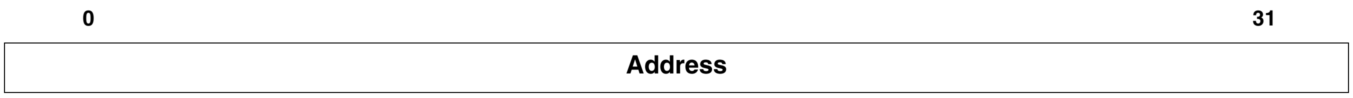


Figure 14-6: **TX Current Buffer Address**

Table 14-54: **Bit Description for TX Current Buffer Address**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	Contains the current payload address. This field changes dynamically when the DMA is operating. This address is a byte address.

Register 0x82, 0x9A, 0xB2, 0xCA – TX Current Buffer Length (TX_CURBUF_LENGTH)

Figure 14-7 shows the TX Current Buffer Length register. Table 14-55 defines the bits in this register.

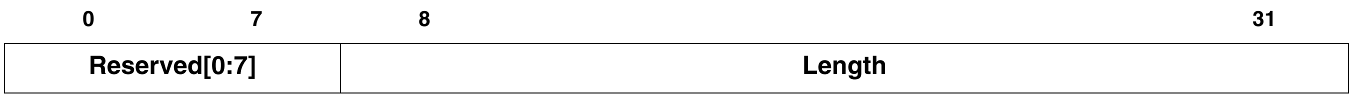


Figure 14-7: **TX Current Buffer Length Register**

Table 14-55: **Bit Description for the TX Current Buffer Length Register**

Bit	Name	Default	Description
[8:31]	Length	`24'h00_0000`	Contains the remaining 24-bit payload length to be transferred. This field changes dynamically when the DMA is operating.
[0:7]	Reserved		

Register 0x83, 0x9B, 0xB3, 0xCB – TX Current Descriptor Pointer (TX_CURDESC_PTR)

Figure 14-8 shows the TX Current Descriptor Pointer register. Table 14-56 defines the bits in this register.

0
31
Address

Figure 14-8: **TX Current Descriptor Pointer**

Table 14-56: **Bit Descriptions for the TX Current Descriptor Pointer**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	Contains the address of the currently executing descriptor. Must be eight-word aligned.

Register 0x84, 0x9C, 0xB4, 0xCC – TX Tail Descriptor Pointer (TX_TAILDESC_PTR)

Figure 14-9 shows the TX Tail Descriptor Pointer register. Table 14-57 defines the bits in this register.

0
31
Address

Figure 14-9: **TX Tail Descriptor Pointer**

Table 14-57: **Bit Descriptions for the TX Tail Descriptor Pointer**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	These bits contain the address of the last descriptor to be fetched. When this register is written to, it initiates a fetch from the address pointed to by the TX Current Descriptor Pointer register. This register can be updated dynamically, while the DMA channel is busy. The control register field, TailPtrEn, must be set for this feature to be enabled.

Register 0x85, 0x9D, 0xB5, 0xCD – TX Channel Control (TX_CHANNEL_CTRL)

Figure 14-10 shows the TX Channel Control register. Table 14-58 defines the bits in this register. This register controls operation for the TX channel only. These registers are initialized by embedded processor block attributes DMA0_TXCHANNELCTRL through DMA3_TXCHANNELCTRL.

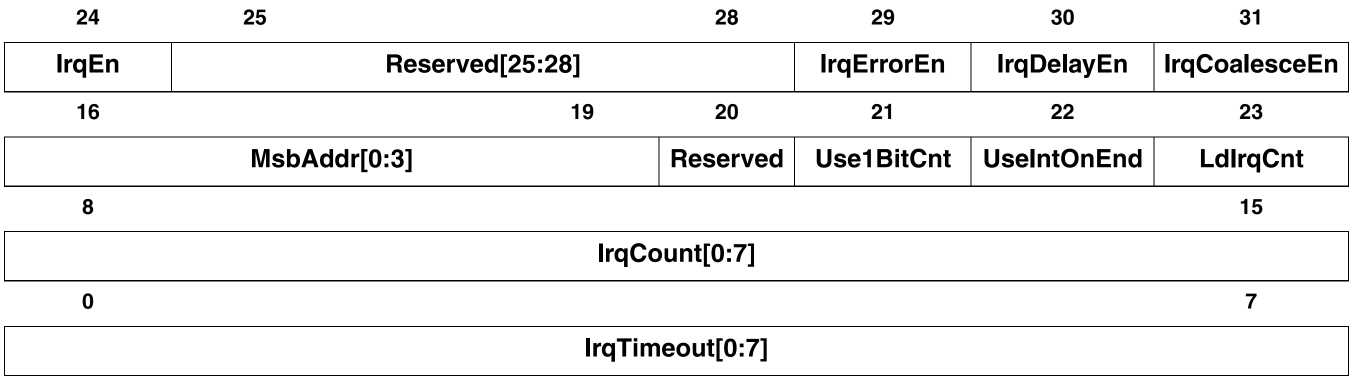


Figure 14-10: **TX Channel Control Register**

Table 14-58: **Bit Descriptions for the TX Channel Control Register**

Bit	Name	Default	Description
[31]	Coalescing Mechanism Interrupt Enable	dmtxchannelctrl[31]	Enable (1) or disable (0) the Coalescing interrupt mechanism.
[30]	Delay Timer Mechanism Interrupt Enable	dmtxchannelctrl[30]	Enable (1) or disable (0) the Delay Timer interrupt mechanism.
[29]	Error Detect Mechanism Interrupt Enable	dmtxchannelctrl[29]	Enable (1) or disable (0) the Error Detection interrupt mechanism.
[25:28]	Reserved	dmtxchannelctrl[25:28]	Reserved
[24]	Master Interrupt Enable	dmtxchannelctrl[24]	When set, this bit indicates that the DMA TX channel is enabled to generate interrupts to the CPU. This is the *master* enable for the TX channel. Individual interrupt sources can be enabled or disabled separately.
[23]	Load the Interrupt Coalescing Counter	`1'b0`	Writing a 1 to this field forces the loading of the Interrupt Coalescing counters from the DCR IrqCount[0:7] field. This bit is self-clearing.
[22]	Use the Interrupt-On-End Mechanism	dmtxchannelctrl[22]	• 1: Select the interrupt-on-end mechanism for interrupt coalescing. • 0: Select the eop mechanism for interrupt coalescing.
[21]	Use 1-bit Interrupt Counters	dmtxchannelctrl[21]	When this bit is enabled, the four-bit Interrupt Coalescing counter and two-bit Delay Timer counters are forced to be one-bit only. For certain device driver applications, this is a desirable use model.
[20]	Reserved	dmtxchannelctrl[20]	Reserved

Table 14-58: **Bit Descriptions for the TX Channel Control Register *(Cont'd)***

Bit	Name	Default	Description
[16:19]	Msb Address	dmtxchannelctrl[16:19]	These bits contain the statically assigned, most-significant four bits of the DMA address. This field must be all zeros.
[8:15]	Interrupt Coalescing Count Value	dmtxchannelctrl[8:15]	These bits contain the eight-bit value to be preloaded into the TX interrupt coalescing counter. They are loaded into the counter when a write to the TX LdIrqCnt field is performed and subsequently reloaded whenever the Count reaches 0.
[0:7]	Interrupt Delay Time-out Value	dmtxchannelctrl[0:7]	These bits hold the compare value for the TX interrupt delay timer. The value in this field is compared to the TX Irq Delay Timer output. When they are equal, a TX interrupt event is generated.

Register 0x86, 0x9E, 0xB6, 0xCE – TX Interrupt Register (TX_IRQ_REG)

This register contains the interrupt status bits for the TX channel as well as read-only status for the TX coalescing and delay timer counters and timers. There are three regular interrupt sources: ErrorIrq, DelayIrq, and CoalesceIrq. There are two non-maskable interrupts (NMI): PlbRdErr and PlbWrErr.

A regular interrupt can be acknowledged (and hence cleared if the corresponding counter equals 0), by writing a "1" to the respective interrupt status bit in this register. The NMIs can only be cleared by issuing a reset to the DMA (hard or soft).

Figure 14-11 shows the TX Interrupt register. Table 14-59 defines the bits in this register.

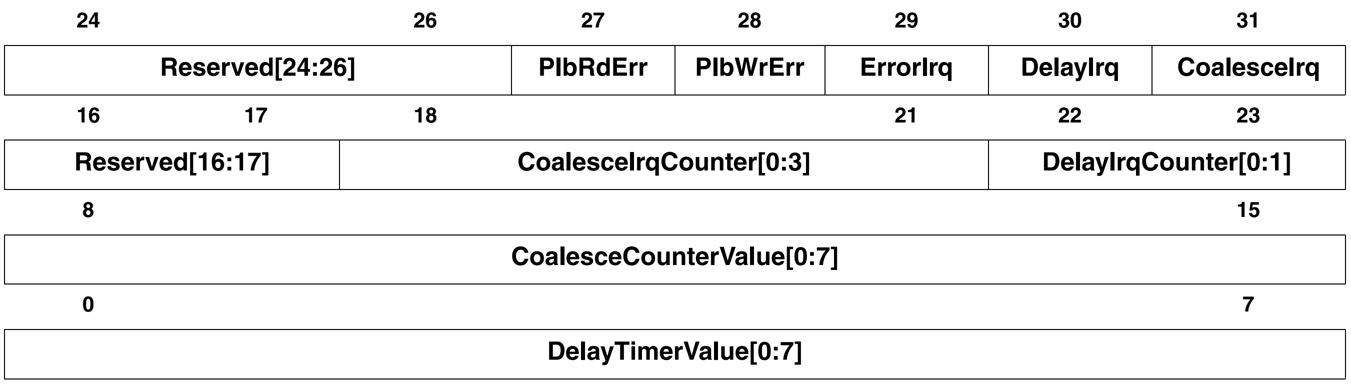


Figure 14-11: **TX Interrupt Register**

Table 14-59: **Bit Descriptions for the TX Interrupt Register**

Bit	Name	Default	Description
[31]	Coalescing Counter Interrupt	`1'b0`	When this bit is 1 the TX DMA channel has a pending interrupt because of a TX Coalescing interrupt counter greater-than-0 condition. This bit is ORed with the two other TX interrupt bits and ANDed with the TX Interrupt Enable bit to produce the TX Irq pin. Even if the TxIrqEn bit is disabled, software can still poll this bit. Acknowledging a TX interrupt due to a coalescing counter condition is accomplished by writing a 1 to this bit. This action decrements the TX Coalescing interrupt counter.
[30]	Delay Timer Interrupt	`1'b0`	When this bit is 1, the TX DMA channel has a pending interrupt because of a TX Delay Timer interrupt counter greater-than-0 condition. This bit is ORed with the two other TX interrupt bits and ANDed with the TX Interrupt Enable bit to produce the TX Irq pin. Even if the TxIrqEn bit is disabled, software can still poll this bit. Acknowledging a TX interrupt due to a Delay Timer counter condition is accomplished by writing a 1 to this bit. This action decrements the TX Delay Timer interrupt counter.
[29]	Error Interrupt	`1'b0`	When this bit is 1, the TX DMA channel has a pending interrupt because of a TX error that has occurred. This bit is ORed with the two other TX interrupt bits and ANDed with the TX Interrupt Enable bit to produce the TX Irq pin. Even if the TxIrqEn bit is disabled, software can still poll this bit. Acknowledging a TX interrupt due to an error is accomplished by writing a 1 to this bit. This action clears this bit.
[28]	PLB Write Error Non-Maskable Interrupt	`1'b0`	When this bit is 1, the TX DMA channel has received an error from the PLB due to a PLB write operation. This serious error causes the DMA to freeze the LocalLink interface as soon as it receives this indication from the crossbar. This bit can only be cleared by resetting the DMA (hard or soft).
[27]	PLB Read Error Non-Maskable Interrupt	`1'b0`	When this bit is 1, the TX DMA channel has received an error from the PLB due to a PLB read operation. This serious error causes the DMA to freeze the LocalLink interface as soon as it receives this indication from the crossbar. This bit can only be cleared by resetting the DMA (hard or soft).
[24:26]	Reserved		
[22:23]	Delay Timer Interrupt Counter		This read-only field is useful for debug purposes. It contains the value of the two-bit counter used to store the number of TX Delay Timer interrupts that are outstanding.
[18:21]	Coalescing Interrupt Counter		This read-only field is useful for debug purposes. It contains the value of the four-bit counter used to store the number of TX coalescing counter interrupts that are outstanding.
[16:17]	Reserved		
[8:15]	Coalescing Counter Value		This read-only field is useful for debug purposes. It contains the value of the eight-bit Coalescing Counter.
[0:7]	Delay Timer Value		This read-only field is useful for debug purposes. It contains the value of the eight-bit Delay Timer.

Register 0x87, 0x9F, 0xB7, 0xCF – TX Status Register (TX_STATUS_REG)

Figure 14-12 shows the TX Status register. Table 14-60 defines the bits in this register. Even though most of these fields are writable via DCR, this is purely for debug purposes. In normal operation, this register should not be directly written.

24	25	26	27	28	29	30	31
Error	**IrqOnEnd**	**StopOnEnd**	**Completed**	**Sop**	**Eop**	**TXChannelBusy**	**Reserved**

16	23
Reserved[16:23]	

8	15
Reserved[8:15]	

0	7
Reserved[0:7]	

Figure 14-12: **TX Status Register**

Table 14-60: **Bit Descriptions for the TX Status Register**

Bit	Name	Default	Description
[31]	Reserved		
[30]	DMA Engine Busy	`1'b0`	When set, this read-only bit indicates that the respective channel is busy with a DMA operation. In general, software should not write any DMA registers while this bit is set. Reading of registers is allowed.
[29]	DMA End of Packet	`1'b0`	When set, this bit indicates that the current descriptor is the final one of a packet. For TX, the CPU sets this bit in the descriptor to indicate that this is the last descriptor of a packet to be transmitted.
[28]	DMA Start of Packet	`1'b0`	When set, this bit indicates that the current descriptor is the start of a packet. For TX, the CPU sets this bit in the descriptor to indicate that this is the first descriptor of a packet to be transmitted.
[27]	DMA Completed	`1'b0`	When set, this bit indicates that the DMA has transferred all data defined by the current descriptor. In the case of TX, the DMA transfers data until the length field specified in the descriptor is zero, and then sets this bit.
[26]	DMA Stop On End	`1'b0`	When this bit is set, the DMA is forced to halt operations when the descriptor is completed. The CPU sets this bit in the status field of the descriptor. This bit is then read into the DMA TX Status register as each descriptor is processed. It is recommended that this bit be set on the EOP descriptor only.

Table 14-60: **Bit Descriptions for the TX Status Register *(Cont'd)***

Bit	Name	Default	Description
[25]	DMA Interrupt on End	`1'b0`	When this bit is set, the DMA is forced to generate an interrupt event when the descriptor is completed. The CPU sets this bit in the status field of the descriptor. This bit is then read into the DMA TX Status register as each descriptor is processed. A typical use model would be to set this bit on the EOP descriptor only. However, it might be set for intermediate descriptors, too. Refer to the UseIntOnEnd field in the TX Channel Control register for details on how to enable this feature.
[24]	DMA Error	`1'b0`	When this bit is set, the DMA encountered a TX error. This bit is a copy of the ErrorIrq bit in the TX Interrupt register.
[0:23]	Reserved		

Register 0x88, 0xA0, 0xB8, 0xD0 – RX Next Descriptor Pointer (RX_NXTDESC_PTR)

Figure 14-13 shows the RX Next Descriptor Pointer register. Table 14-61 defines the bits in this register.

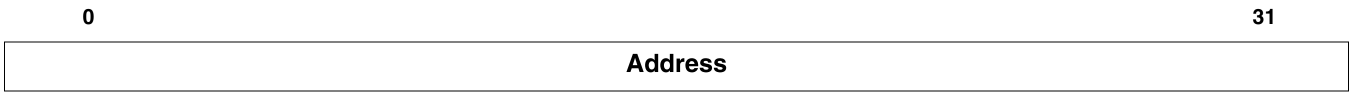


Figure 14-13: **RX Next Descriptor Pointer**

Table 14-61: **Bit Descriptions for the RX Next Descriptor Pointer**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	Contains the address of the next descriptor to be fetched. Must be eight-word aligned.

Register 0x89, 0xA1, 0xB9, 0xD1 – RX Current Buffer Address (RX_CURBUF_ADDR)

Figure 14-14 shows the RX Current Buffer Address register. Table 14-62 defines the bits in this register.

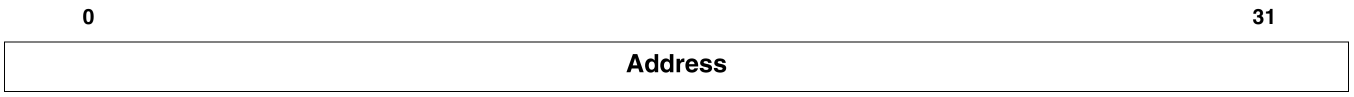


Figure 14-14: **RX Current Buffer Address Register**

Table 14-62: **Bit Descriptions for the RX Current Buffer Address Register**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	Contains the current payload address. This field changes dynamically when the DMA is operating. This address is a byte address.

Register 0x8A, 0xA2, 0xBA, 0xD2 – RX Current Buffer Length (RX_CURBUF_LENGTH)

Figure 14-15 shows the RX Current Buffer Length register. Table 14-63 defines the bits in this register.

0 7	8 31
Reserved[0:7]	**Length**

Figure 14-15: **RX Current Buffer Length Register**

Table 14-63: **Bit Descriptions for the RX Current Buffer Length Register**

Bit	Name	Default	Description
[8:31]	Length	`24'h00_0000`	Contains the remaining 24-bit payload length to be transferred. This field changes dynamically when the DMA is operating.
[0:7]	Reserved		

Register 0x8B, 0xA3, 0xBB, 0xD3 – RX Current Descriptor Pointer (RX_CURDESC_PTR)

Figure 14-16 shows the RX Current Descriptor Pointer register. Table 14-64 defines the bits in this register.

0 31
Address

Figure 14-16: **RX Current Descriptor Pointer**

Table 14-64: **Bit Descriptions for the RX Current Descriptor Pointer**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	Contains the address of the currently executing descriptor. Must be eight-word aligned.

Register 0x8C, 0xA4, 0xBC, 0xD4 – RX Tail Descriptor Pointer (RX_TAILDESC_PTR)

Figure 14-17 shows the RX Tail Descriptor Pointer register. Table 14-65 defines the bits in this register.

0 31
Address

Figure 14-17: **RX Tail Descriptor Pointer**

Table 14-65: **Bit Descriptions for the RX Tail Descriptor Pointer**

Bit	Name	Default	Description
[0:31]	Address	`32'h0000_0000`	Contains the address of the last descriptor to be fetched. When this register is written to, it initiates a fetch from the address pointed to by the RX Current Descriptor Pointer register. This register can be updated dynamically, while the DMA channel is busy. The control register field, TailPtrWrEn, must be set for this feature to be enabled.

Register 0x8D, 0xA5, 0xBD, 0xD5 – RX Channel Control (RX_CHANNEL_CTRL)

Figure 14-18 shows the RX Channel Control register. Table 14-66 defines the bits in this register. This register controls operation for the RX channel only. These registers are initialized by embedded processor block attributes DMA0_RXCHANNELCTRL through DMA3_RXCHANNELCTRL.

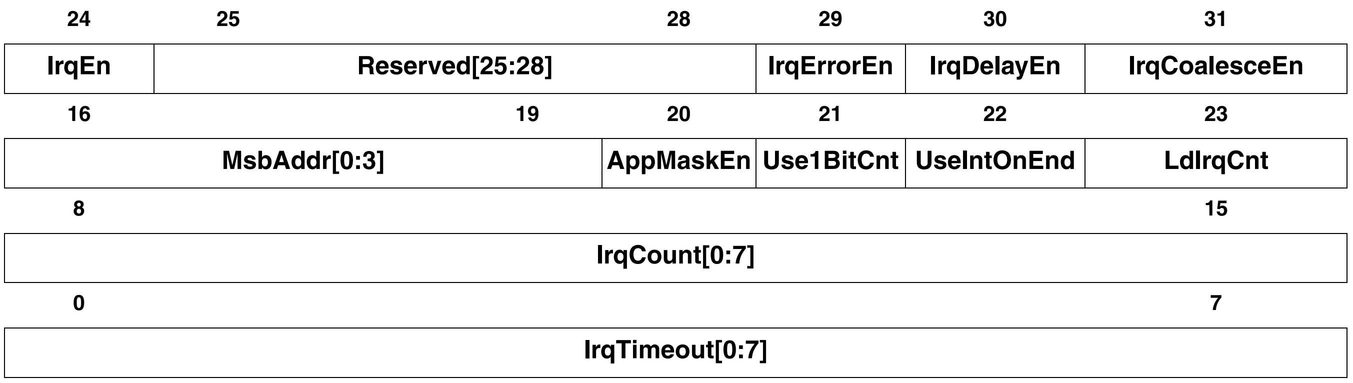


Figure 14-18: **RX Channel Control Register**

Table 14-66: **Bit Descriptions for the RX Channel Control Register**

Bit	Name	Default	Description
[31]	Coalescing Mechanism Interrupt Enable	dmrxchannelctrl[31]	Enable (1) or disable (0) the Coalescing interrupt mechanism.
[30]	Delay Timer Mechanism Interrupt Enable	dmrxchannelctrl[30]	Enable (1) or disable (0) the Delay Timer interrupt mechanism.
[29]	Error Detect Mechanism Interrupt Enable	dmrxchannelctrl[29]	Enable (1) or disable (0) the Error Detection interrupt mechanism.
[25:28]	Reserved	dmrxchannelctrl[25:28]	Reserved Bits.
[24]	Master Interrupt Enable	dmrxchannelctrl[24]	When this bit is set, the DMA RX channel is enabled to generate interrupts to the CPU. This is the *master* enable for the RX channel. Individual interrupt sources can be enabled or disabled separately.
[23]	Load the Interrupt Coalescing Counter	`1'b0`	Writing a 1 to this bit forces the loading of the Interrupt Coalescing counters from the DCR IrqCount[0:7] field. This bit is self-clearing.

Table 14-66: **Bit Descriptions for the RX Channel Control Register *(Cont'd)***

Bit	Name	Default	Description
[22]	Use the Interrupt-On-End Mechanism	dmrxchannelctrl[22]	• 1: Select the interrupt-on-end mechanism for interrupt coalescing. • 0: Select the eop mechanism for interrupt coalescing.
[21]	Use 1-bit Interrupt Counters	dmtxchannelctrl[21]	When this bit is enabled, the four-bit Interrupt Coalescing counter and two-bit Delay Timer counters are forced to be one-bit only. For certain device driver applications, this is a desirable use model.
[20]	Application Data Mask Enable	dmrxchannelctrl[20]	This bit enables the Application Data Mask mode. Refer to "Masking of Application Data Update," page 235 for details of operation.
[16:19]	Msb Address	dmtxchannelctrl[16:19]	These bits contain the statically assigned, most-significant four bits of the DMA address. This field must be all zeros.
[8:15]	Interrupt Coalescing Count Value	dmrxchannelctrl[8:15]	These bits contain the eight-bit value to be preloaded into the RX interrupt coalescing counter. This value is loaded into the counter when a write to the RX LdIrqCnt field is performed and subsequently reloaded whenever the Count reaches 0.
[0:7]	Interrupt Delay Time-out Value	dmrxchannelctrl[0:7]	These bits hold the compare value for the RX interrupt delay timer. The value in this register is compared to the RX Irq Delay Timer output. When they are equal, an RX interrupt event is generated.

Register 0x8E, 0xA6, 0xBE, 0xD6 – RX Interrupt Register (RX_IRQ_REG)

This register contains the interrupt status bits for the RX channel as well as the read-only status for the RX coalescing and Delay timer counters. There are three regular interrupt sources: ErrorIrq, DelayIrq, and CoalesceIrq. There are two non-maskable interrupts (NMI): PlbRdErr and PlbWrErr. A regular interrupt can be acknowledged (and hence cleared if the corresponding counter equals 0) by writing a "1" to the respective interrupt status bit in this register. The NMIs can only be cleared by issuing a reset to the DMA (hard or soft).

Figure 14-19 shows the RX Interrupt register. Table 14-67 defines the bits in this register.

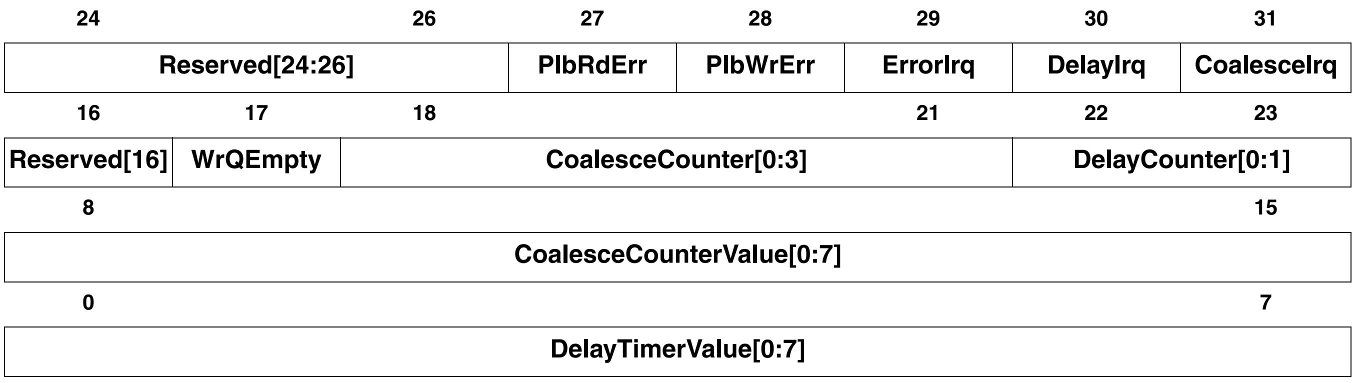


Figure 14-19: **RX Interrupt Register**

Table 14-67: **Bit Descriptions for the RX Interrupt Register**

Bit	Name	Default	Description
[31]	Coalescing Counter Interrupt	`1'b0`	When this bit is 1, the RX DMA channel has a pending interrupt because of an RX Coalescing interrupt counter greater-than-0 condition. This bit is ORed with the two other RX interrupt bits and ANDed with the RX Interrupt Enable bit to produce the RX Irq pin. Even if the RxIrqEn or IrqCoalesceEn bit is disabled, software can still poll this bit. Acknowledging an RX interrupt due to a coalescing counter condition is accomplished by writing a 1 to this bit. This action decrements the RX Coalescing interrupt counter.
[30]	Delay Timer Interrupt	`1'b0`	When this bit is 1, the RX DMA channel has a pending interrupt because of an RX Delay Timer interrupt counter greater-than-0 condition. This bit is ORed with the two other RX interrupt bits and ANDed with the RX Interrupt Enable bit to produce the RX Irq pin. Even if the RxIrqEn or IrqDelayEn bit is disabled, software can still poll this bit. Acknowledging an RX interrupt due to a Delay Timer counter condition is accomplished by writing a 1 to this bit. This action decrements the RX Delay Timer interrupt counter.
[29]	Error Interrupt	`1'b0`	When this bit is 1, the RX DMA channel has a pending interrupt because of an RX error that has occurred. This bit is ORed with the two other RX interrupt bits and ANDed with the RX Interrupt Enable bit to produce the RX Irq pin. Even if the RxIrqEn or IrqErrorEn bit is disabled, software can still poll this bit. Acknowledging an RX interrupt due to an Error is accomplished by writing a 1 to this bit. This action clears this bit.
[28]	PLB Write Error Non-Maskable Interrupt	`1'b0`	When this bit is 1, the RX DMA channel has received an error from the PLB due to a PLB write operation. This serious error causes the DMA to freeze the LocalLink interface as soon as it receives this indication from the crossbar. This bit can only be cleared by resetting the DMA (hard or soft).
[27]	PLB Read Error Non-Maskable Interrupt	`1'b0`	When this bit is 1, the RX DMA channel has received an error from the PLB due to a PLB read operation. This serious error causes the DMA to freeze the LocalLink interface as soon as it receives this indication from the crossbar. This bit can only be cleared by resetting the DMA (hard or soft).
[24:26]	Reserved		
[22:23]	Delay Timer Interrupt Counter		This read-only field is useful for debug purposes. It contains the value of the two-bit counter used to store the number of RX Delay Timer interrupts that are outstanding.
[18:21]	Coalescing Interrupt Counter		This read-only field is useful for debug purposes. It contains the value of the four-bit counter used to store the number of RX coalescing counter interrupts that are outstanding.
[17]	Write Command Queue Empty Status		This read-only field is useful for debug purposes. It indicates whether the Write Command Queue is empty (1) or not (0). If the DMA is paused, reading this field indicates that all the write data associated with the pending commands has been flushed.

Table 14-67: **Bit Descriptions for the RX Interrupt Register *(Cont'd)***

Bit	Name	Default	Description
[16:]	Reserved		
[8:15]	Coalescing Counter Value		This read-only field is useful for debug purposes. It contains the value of the eight-bit Coalescing Counter.
[0:7]	Delay Timer Value		This read-only field is useful for debug purposes. It contains the value of the eight-bit Delay Timer.

Register 0x8F, 0xA7, 0xBF, 0xD7 – RX Status Register (RX_STATUS_REG)

Figure 14-20 shows the RX Status register. Table 14-68 defines the bits in this register. Even though most of these fields are writable via DCR, this register is purely for debug purposes. In normal operation, this register should not be directly written.

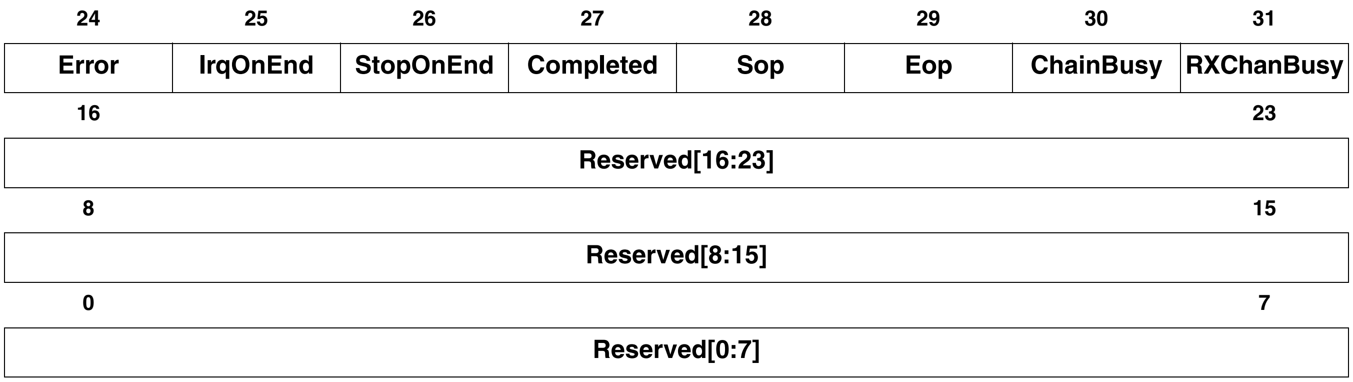


Figure 14-20: **RX Status Register**

Table 14-68: **Bit Descriptions for the RX Status Register**

Bit	Name	Default	Description
[31]	Reserved		
[30]	DMA Engine Busy	`1'b0`	When this read-only bit is set, the respective channel is busy with a DMA operation. In general, software should not write any DMA registers while this bit is set. Reading of registers is allowed.
[29]	DMA End of Packet	`1'b0`	When this bit is set, the current descriptor is the final one of a packet. For RX, when an EOP is received by the LocalLink interface, the DMA sets this bit in the descriptor to inform the CPU that the current descriptor is the last of a received packet.
[28]	DMA Start of Packet	`1'b0`	When this bit is set, the current descriptor is the start of a packet. For RX, when an SOP is received by the LocalLink interface, the DMA sets this bit in the descriptor to inform the CPU that the current descriptor is the first of a received packet.
[27]	DMA Completed	`1'b0`	When this bit is set, the DMA has transferred all data defined by the current descriptor. For RX, the DMA transfers data until the length field specified in the descriptor is zero OR when it receives an EOP indication from the LocalLink interface. At that point, the DMA sets this bit.

Table 14-68: **Bit Descriptions for the RX Status Register *(Cont'd)***

Bit	Name	Default	Description
[26]	DMA Stop On End	`1'b0`	When this bit is set, the DMA is forced to halt operations when the descriptor is completed. The CPU sets this bit in the status field of the descriptor. This bit is then read into the DMA RX Status register as each descriptor is processed. It is recommended that this bit be set, corresponding to an EOP descriptor only.
[25]	DMA Interrupt on End	`1'b0`	When this bit is set, the DMA is forced to generate an interrupt event when the descriptor is completed. The CPU sets this bit in the status field of the descriptor. This bit is then read into the DMA RX Status register as each descriptor is processed. A typical use model is to set this bit on the EOP descriptor only. However, it can be set for intermediate descriptors, too. Refer to the UseIntOnEnd field in the RX Channel Control register for details of how to enable this feature.
[24]	DMA Error	`1'b0`	When this bit is set, the DMA encountered an RX error. This bit is a copy of the ErrorIrq bit in the RX Interrupt register.
[0:23]	Reserved		

Register 0x90, 0xA8, 0xC0, 0xD8 – DMA Control Register (DMA_CONTROL_REG)

Figure 14-21 shows the DMA Control register. Table 14-69 defines the bits in this register. This register contains control fields that affect both the RX and TX channels. These registers are initialized by embedded processor block attributes DMA0_CONTROL through DMA3_CONTROL.

24	25	26	27	28	29	30	31
Reserved[24:25]		PlbErrDisable	OverFlowErrDisable[0:1]		TailPtrEn	Reserved	SwReset

0	23
Reserved[0:23]	

Figure 14-21: **DMA Control Register**

Table 14-69: **Bit Descriptions for the DMA Control Register**

Bit	Name	Description
[31]	Software Reset	Writing a 1 to this bit forces the DMA engine (both RX and TX channels) to shut down and reset itself. Because the dma_ll_rst_engine_ack output is asserted when this bit is a 1, it can be used to reset a remote LocalLink device while the DMA engine is resetting itself. After setting this bit, software must poll it until the bit is cleared by the DMA, which indicates that the reset process is done and the pipeline has been flushed.
[30]	Reserved	
[29]	Tail Pointer Enable	When this bit is set, the Tail Pointer mechanism is enabled. In this mode, writing to the tail pointer initiates a DMA transaction and the comparison (tail pointer == current pointer) ends descriptor execution. When cleared, the legacy mode of writing to the current pointer to initiate a transfer is supported. Refer to "DMA Legacy Mode," page 231 for details.

Table 14-69: **Bit Descriptions for the DMA Control Register *(Cont'd)***

Bit	Name	Description
[27:28]	Overflow Counter Error Interrupt Disable	When this bit is set, the error interrupt is disabled when either the two-bit Delay Timer counter or the four-bit Coalescing counter overflows. Bit [27] is used for the RX channel, and Bit [28] is used for the TX channel.
[26]	PLB Error Disable	When this bit is set, error checking is disabled due to reads/writes to and from the crossbar PLB. If one of these errors occurs, the DMA reacts as follows: • PLB Error Disable = `1'b1`: The DMA ignores the error and continues as usual. • PLB Error Disable = `1'b0`: • Read Data Error. The DMA logs a plb_rd_error NMI bit in the appropriate interrupt register (RX or TX). The LocalLink interface is frozen immediately. • Write Data Error. The DMA logs a plb_wr_error NMI bit in both RX and TX interrupt registers. The LocalLink interfaces is frozen immediately.
[0:25]	Reserved	

Chapter 15

APU Programming

Introduction

This chapter describes the Xilinx extension of the PowerPC 440 ISA through the Auxiliary Processor Unit (APU) of the embedded processor block in Virtex-5 FXT FPGAs. The processor relies on custom Fabric Coprocessing Modules (FCMs) implemented in the FPGA logic to execute these instructions. An FCM can be either a user IP or a Xilinx IP.

This chapter lists the APU instruction set extension supported by the processor on Virtex-5 FXT devices.

The notation **FCM5** in this document indicates a five-bit immediate value. The interpretation of the value is left to the FCM. Typically, this is the register value on the FCM.

Refer to the "Document Convention" section of the *PowerPC Processor Reference Guide* for definitions of the remaining notations.

If there is an unconnected FPU in hardware, the APU and the Disable FPU Decode bits in the APU Control register must not be enabled by software because an FP unavailable exception will result. This condition implies that software can never enable the APU and clear the Disable FPU Decode bits in the APU Control register unless the designer knows whether a FPU is connected or not. Tools or the user must configure the APU control register properly. Without an FPU, the Disable FPU Decode bits should be set.

udi<*n*>fcm
udi<*n*>fcm.

User-Defined Instructions (UDIs)

udi<*n*>fcm	T, A, B	UDIs that do not modify the condition code register
udi<*n*>fcm.	T, A, B	UDIs that modify the condition code register

4	T	A	B	1	n	3	Rcn
0 5	6 10	11 15	16 20	21	22 25	26 30	31

Description

The exact operation done by the instruction is determined by the user based on the FCM. For more information about user-defined instructions, refer to "FCM User-Defined Instructions," page 196.

The Xilinx GNU assembler recognizes 32 UDIs:

- 16 instructions that modify the condition code (**Rcn** = 0)
- 16 instructions that do not modify the condition code (**Rcn** = 1)

Note that <n> in the mnemonic can range from 0 to 15. The instruction definition as provided by the user determines which conditional code register is to be modified.

Special Notations for this Instruction

```
T : PowerPC GPR (rD)/ FCM Register (FCR)
A : PowerPC GPR (rD)/ FCM Register (FCR)/ 5 bit Immediate
B : PowerPC GPR (rD)/ FCM Register (FCR)/ 5 bit Immediate
```

Five bits each are available for the special notations defined above.

Pseudocode

Dependent on the user operation.

Registers Altered

Dependent on the user operation.

Exceptions

Dependent on the user operation.

Compatibility

These instructions are defined by Xilinx as UDIs that use the APU controller. The processor in the embedded processor block in Virtex-5 FXT FPGAs allows for 16 UDIs as compared to 8 allowed by the PowerPC 405 processor in Virtex-4 devices.

lbfcmux

Load Byte with Update Indexed (FCM)

lbfcmux **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	519	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address.

The byte referenced by EA is sign-extended to 32 bits and loaded into the **FCM5** register. The EA is loaded into **r**A.

Pseudocode

```
EA                          _ (rA) + (rB)
eb                          _ 8 * $EA_{28:31}$
APU(FCM).data               _ undefined
$APU(FCM).data_{eb:eb+7}$   _ MEM(EA,1)
(FCM5)                      _ custom_op(APU(FCM).data, (FCM5))
(rA)                        _ EA
```

Registers Altered

- **r**A
- Register inferred by **FCM5**

Exceptions

- Data storage: this exception is raised if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value
- **r**A = 0

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

lhfcmux

Load Halfword with Update Indexed (FCM)

lhfcmux **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	551	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address.

The halfword referenced by EA is sign-extended to 32 bits and loaded into the **FCM5** register. The EA is loaded into **r**A.

Pseudocode

```
EA                          _ ((rA) + (rB)) & (~1)
eb                          _ 8 * $EA_{28:31}$
APU(FCM).data               _ undefined
$APU(FCM).data_{eb:eb+15}$    _ MEM(EA,2)
(FCM5)                      _custom_op(APU(FCM).data, (FCM5))
(rA)                        _ EA
```

Registers Altered

- **r**A
- Register inferred by **FCM5**

Exceptions

- Data storage: this exception is raised if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value
- **r**A = 0

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

lwfcmux

Load Word With Update Indexed (FCM)

lwfcmux **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	583	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address.

The word referenced by EA is loaded into the **FCM5** register. The EA is loaded into **r**A.

Pseudocode

```
EA                       _ ((rA) + (rB)) & (~3)
eb                       _ 8 * EA$_{28:31}$
APU(FCM).data            _ MEM(EA,4)
(FCM5)                   _ custom_op(APU(FCM).data, (FCM5))
(rA)                     _ EA
```

Registers Altered

- **r**A
- Register inferred by **FCM5**

Exceptions

- Data storage: this exception is raised if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value
- **r**A = 0

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

ldfcmux

Load Double with Update Indexed (FCM)

ldfcmux **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	775	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address.

Two words referenced by EA and EA +4 are loaded into the **FCM5** register. The EA is loaded into **r**A.

Pseudocode

```
EA                      _ ((rA) + (rB)) & (~3)
eb                      _ 8 * EA$_{28:31}$
APU(FCM).data           _ MEM(EA,4)
(FCM5)                  _ custom_op(APU(FCM).data, (FCM5))
APU(FCM).data           _ MEM(EA+4,4)
(FCM5)                  _custom_op(APU(FCM).data, (FCM5))
(rA)                    _ EA
```

Registers Altered

- **r**A
- Register inferred by **FCM5**

  The instruction assumes that the **FCM5** register is 64 bits wide.

Exceptions

- Data storage: this exception is raised if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value
- **r**A = 0

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

lqfcmux

Load Quad with Update Indexed (FCM)

lqfcmux **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	615	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address.

Four words referenced by EA through EA + 12 are loaded into the **FCM5** register. The EA is loaded into **r**A.

Pseudocode

```
EA                       _ ((rA) + (rB)) & (~3)
eb                       _ 8 * $EA_{28:31}$
APU(FCM).data            _ MEM(EA,4)
(FCM5)                   _ custom_op(APU(FCM).data, (FCM5))
APU(FCM).data            _ MEM(EA+4,4)
(FCM5)                   _ custom_op(APU(FCM).data, (FCM5))
APU(FCM).data            _ MEM(EA+8,4)
(FCM5)                   _custom_op(APU(FCM).data, (FCM5))
APU(FCM).data            _ MEM(EA+12,4)
(FCM5)                   _ custom_op(APU(FCM).data, (FCM5))
(rA)                     _ EA
```

Registers Altered

- **r**A
- Register inferred by **FCM5**

  The instruction assumes that the **FCM5** register is 128 bits wide.

Exceptions

- Data storage: this exception is raised if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value
- **r**A = 0

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

lbfcmx

Load Byte Indexed (FCM)

lbfcmx **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	7	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address. If **r**A is 0, then 0 is used as the base address.

The byte referenced by EA is sign-extended to 32 bits and loaded into the **FCM5** register.

Pseudocode

```
EA                          _ (0|rA) + (rB)
eb                          _ 8 * EA$_{28:31}$
APU(FCM).data               _ undefined
APU(FCM).data$_{eb:eb+7}$       _ MEM(EA,1)
(FCM5)                      _ custom_op(APU(FCM).data, (FCM5))
```

Registers Altered

Register inferred by **FCM5**

Exceptions

- Data storage: this exception is raised if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

lhfcmx

Load Halfword Indexed (FCM)

lhfcmx **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	39	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address. If **r**A is 0, then 0 is used as the base address.

The halfword referenced by EA is sign-extended to 32 bits and loaded into the **FCM5** register.

Pseudocode

```
EA                              _ ((0|rA) + (rB)) & (~1)
eb                              _ 8 * EA$_{28:31}$
APU(FCM).data                   _ undefined
APU(FCM).data$_{eb:eb+15}$      _MEM(EA,2)
(FCM5)                          _ custom_op(APU(FCM).data, (FCM5))
```

Registers Altered

Register inferred by **FCM5**

Exceptions

- Data storage: this exception is raised if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

lwfcmx

Load Word Indexed (FCM)

lwfcmx **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	71	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address. If **r**A is 0, then 0 is used as the base address.

The word referenced by EA is loaded into the **FCM5** register.

Pseudocode

```
EA                      _ ((0|rA) + (rB)) & (~3)
eb                      _ 8 * $EA_{28:31}$
APU(FCM).data           _ MEM(EA,4)
(FCM5)                  _ custom_op(APU(FCM).data, (FCM5))
```

Registers Altered

Register inferred by **FCM5**

Exceptions

- Data storage: this exception is raised if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

ldfcmx

Load Double Indexed (FCM)

ldfcmx **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	263	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address. If **r**A is 0, then 0 is used as the base address.

Two words referenced by EA and EA + 4 are loaded into register(s) inferred by **FCM5**.

Pseudocode

```
EA                        _ ((0|rA) + (rB)) & (~3)
eb                        _ 8 * $EA_{28:31}$
APU(FCM).data             _ MEM(EA,4)
(FCM5)                    _ custom_op(APU(FCM).data, (FCM5))
APU(FCM).data             _ MEM(EA+4,4)
(FCM5)                    _ custom_op(APU(FCM).data, (FCM5))
```

Registers Altered

Register inferred by **FCM5**. The exact implementation of the register inferred is to be determined by the user.

Exceptions

- Data storage: this exception is raised if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

lqfcmx

Load Quad Indexed (FCM)

lqfcmx FCM5, rA, rB

31	FCM5	rA	rB	103	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address. If **r**A is 0, then 0 is used as the base address.

Four words referenced by EA through EA + 12 are loaded into register(s) inferred by **FCM5**.

Pseudocode

```
EA                  _ ((0|rA) + (rB)) & (~3)
eb                  _ 8 * $EA_{28:31}$
APU(FCM).data       _ MEM(EA,4)
(FCM5)              _custom_op(APU(FCM).data, (FCM5))
APU(FCM).data       _ MEM(EA+4,4)
(FCM5)              _ custom_op(APU(FCM).data, (FCM5))
APU(FCM).data       _ MEM(EA+8,4)
(FCM5)              _ custom_op(APU(FCM).data, (FCM5))
APU(FCM).data       _ MEM(EA+12,4)
(FCM5)              _ custom_op(APU(FCM).data, (FCM5))
```

Registers Altered

Register inferred by **FCM5**. The exact implementation of the register inferred is to be determined by the user.

Exceptions

- Data storage: this exception is raised if the access is prevented by no-access-allowed zone protection. This only applies to accesses in user mode when data relocation is enabled.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

stbfcmux

Store Byte with Update Indexed (FCM)

stbfcmux **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	647	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address. If **r**A is 0, then 0 is used as the base address.

The least-significant byte of the register inferred by **FCM5** is stored into the byte referenced by EA. The EA is loaded into the **r**A register.

Pseudocode

```
EA                      _ (rA) + (rB)
eb                      _ 8 * EA$_{28:31}$
APU(FCM).data           _ custom_op(FCM5)
MEM(EA,1)               _APU(FCM).data$_{eb:eb+7}$
(rA)                    _ EA
```

Registers Altered

rA

Exceptions

- Data storage: this exception is raised if the access is prevented by zone protection when data relocation is enabled.
  - No-access-allowed zone protection applies only to accesses in user mode.
  - Read-only zone protection applies to user and privileged modes.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value
- **r**A = 0

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

sthfcmux

Store Halfword With Update Indexed (FCM)

stbfcmux **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	679	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address. If **r**A is 0, then 0 is used as the base address.

Two least-significant bytes of the register inferred by **FCM5** are stored into the addressed referenced by EA. The EA is loaded into the **r**A register.

Pseudocode

```
EA                        _ ((rA) + (rB)) & (~1)
eb                        _ 8 * $EA_{28:31}$
APU(FCM).data             _ custom_op(FCM5)
MEM(EA,2)                 _ APU(FCM).$data_{eb:eb+15}$
(rA)                      _EA
```

Registers Altered

rA

Exceptions

- Data storage: this exception is raised if the access is prevented by zone protection when data relocation is enabled.
  - No-access-allowed zone protection applies only to accesses in user mode.
  - Read-only zone protection applies to user and privileged modes.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value
- **r**A = 0

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

stwfcmux

Store Word With Update Indexed (FCM)

stwfcmux **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	711	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address. If **r**A is 0, then 0 is used as the base address.

The contents of the register inferred by **FCM5** are stored into the address referenced by EA. The EA is loaded into the **r**A register.

Pseudocode

```
EA                        _ ((rA) + (rB)) & (~3)
eb                        _ 8 * EA$_{28:31}$
APU(FCM).data             _ custom_op(FCM5)
MEM(EA,4)                 _ APU(FCM).data
(rA)                      _ EA
```

Registers Altered

rA

Exceptions

- Data storage: this exception is raised if the access is prevented by zone protection when data relocation is enabled.
  - No-access-allowed zone protection applies only to accesses in user mode.
  - Read-only zone protection applies to user and privileged modes.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value
- **r**A = 0

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

stdfcmux

Store Double With Update Indexed (FCM)

stdfcmux **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	903	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address. If **r**A is 0, then 0 is used as the base address.

The contents of the register inferred by **FCM5** are stored into the address referenced by EA and EA + 4. The source register is assumed to be 64 bits wide. The EA is loaded into the **r**A register.

Pseudocode

```
EA                  _ ((rA) + (rB)) & (~3)
eb                  _ 8 * EA28:31
APU(FCM).data       _ custom_op(FCM5)
MEM(EA,4)           _ APU(FCM).data
APU(FCM).data       _ custom_op(FCM5)
MEM(EA+4,4)         _ APU(FCM).data
(rA)                _ EA
```

Registers Altered

rA

Exceptions

- Data storage: this exception is raised if the access is prevented by zone protection when data relocation is enabled.
    - No-access-allowed zone protection applies only to accesses in user mode.
    - Read-only zone protection applies to user and privileged modes.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value
- **r**A = 0

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

stqfcmux

Store Quad With Update Indexed (FCM)

stqfcmux **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	743	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address. If **r**A is 0, then 0 is used as the base address.

The contents of the register inferred by **FCM5** are stored into the address referenced by EA through EA+12. The source register is assumed to be 128 bits wide. The EA is loaded into the **r**A register.

Pseudocode

```
EA                    _ ((rA) + (rB)) & (~3)
eb                    _ 8 * $EA_{28:31}$
APU(FCM).data         _ custom_op(FCM5)
MEM(EA,4)             _ APU(FCM).data
APU(FCM).data         _ custom_op(FCM5)
MEM(EA+4,4)           _ APU(FCM).data
APU(FCM).data         _ custom_op(FCM5)
MEM(EA+8,4)           _ APU(FCM).data
APU(FCM).data         _ custom_op(FCM5)
MEM(EA+12,4)          _ APU(FCM).data
(rA)                  _ EA
```

Registers Altered

rA

Exceptions

- Data storage: this exception is raised if the access is prevented by zone protection when data relocation is enabled.
  - No-access-allowed zone protection applies only to accesses in user mode.
  - Read-only zone protection applies to user and privileged modes.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value
- **r**A = 0

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

stbfcmx

Store Byte Indexed (FCM)

stbfcmx **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	135	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address. If **r**A is 0, then 0 is used as the base address.

The least-significant byte of the register inferred by **FCM5** is stored into the byte referenced by EA.

Pseudocode

```
EA                 _ (0|rA) + (rB)
eb                 _ 8 * EA$_{28:31}$
APU(FCM).data      _ custom_op(FCM5)
MEM(EA,1)          _ APU(FCM).data$_{eb:eb+7}$
```

Registers Altered

None

Exceptions

- Data storage: this exception is raised if the access is prevented by zone protection when data relocation is enabled.
  - No-access-allowed zone protection applies only to accesses in user mode.
  - Read-only zone protection applies to user and privileged modes.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

sthfcmx

Store Halfword Indexed (FCM)

sthfcmx **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	167	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address. If **r**A is 0, then 0 is used as the base address.

The two least-significant bytes of the register inferred by **FCM5** are stored into the address referenced by EA.

Pseudocode

```
EA                      _ ((0|rA) + (rB)) & (~1)
eb                      _ 8 * $EA_{28:31}$
APU(FCM).data           _ custom_op(FCM5)
MEM(EA,2)               _APU(FCM).$data_{eb:eb+15}$
```

Registers Altered

None

Exceptions

- Data storage: this exception is raised if the access is prevented by zone protection when data relocation is enabled.
  - No-access-allowed zone protection applies only to accesses in user mode.
  - Read-only zone protection applies to user and privileged modes.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

stwfcmx

Store Word Indexed (FCM)

stwfcmx **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	199	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address. If **r**A is 0, then 0 is used as the base address.

The contents of the register inferred by **FCM5** are stored into the address referenced by EA.

Pseudocode

```
EA                       _ ((0|rA) + (rB)) & (~3)
eb                       _ 8 * $EA_{28:31}$
APU(FCM).data            _ custom_op(FCM5)
MEM(EA,4)                _ APU(FCM).data
```

Registers Altered

None

Exceptions

- Data storage: this exception is raised if the access is prevented by zone protection when data relocation is enabled.
  - No-access-allowed zone protection applies only to accesses in user mode.
  - Read-only zone protection applies to user and privileged modes.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

stdfcmx

Store Double Indexed (FCM)

stdfcmx **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	391	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address. If **r**A is 0, then 0 is used as the base address.

The contents of the register inferred by **FCM5** are stored into the address referenced by EA. The source register is expected to be 64 bits wide.

Pseudocode

```
EA                       _ ((0|rA) + (rB)) & (~3)
eb                       _ 8 * EA$_{28:31}$
APU(FCM).data            _ custom_op(FCM5)
MEM(EA,4)                _ APU(FCM).data
APU(FCM).data            _ custom_op(FCM5)
MEM(EA+4,4)              _ APU(FCM).data
```

Registers Altered

None

Exceptions

- Data storage: this exception is raised if the access is prevented by zone protection when data relocation is enabled.
  - No-access-allowed zone protection applies only to accesses in user mode.
  - Read-only zone protection applies to user and privileged modes.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

stqfcmx

Store Quad Indexed (FCM)

stqfcmx **FCM5**, **r**A, **r**B

31	FCM5	rA	rB	231	0
0 5	6 10	11 15	16 20	21 30	31

Description

An effective address (EA) is calculated by adding an index to a base address, which is formed as follows:

- The contents of the **r**B register are used as the index.
- The contents of the **r**A register are used as the base address. If **r**A is 0, then 0 is used as the base address.

The contents of the register inferred by **FCM5** are stored into the address referenced by EA. The source register is expected to be 128 bits wide.

Pseudocode

```
EA                          _ ((0|rA) + (rB)) & (~3)
eb                          _ 8 * $EA_{28:31}$
APU(FCM).data               _ custom_op(FCM5)
MEM(EA,4)                   _ APU(FCM).data
APU(FCM).data               _ custom_op(FCM5)
MEM(EA+4,4)                 _ APU(FCM).data
APU(FCM).data               _ custom_op(FCM5)
MEM(EA+8,4)                 _ APU(FCM).data
APU(FCM).data               _ custom_op(FCM5)
MEM(EA+12,4)                _ APU(FCM).data
```

Registers Altered

None

Exceptions

- Data storage: this exception is raised if the access is prevented by zone protection when data relocation is enabled.
  - No-access-allowed zone protection applies only to accesses in user mode.
  - Read-only zone protection applies to user and privileged modes.
- Data TLB miss: this exception is raised if data relocation is enabled and a valid translation entry corresponding to the EA is not found in the TLB.

Execution of any of the following invalid instruction forms results in a boundedly undefined result rather than a program exception:

- Reserved bits containing a non-zero value

Compatibility

This instruction is predefined by Xilinx as using the APU controller.

Chapter 16

Additional Programming Considerations

This chapter contains additional information for programmers. The information here overrides what is specified in the *PPC440x5 CPU Core User's Manual* [Ref 5] for PowerPC implementations in Virtex-5 FXT FPGAs.

Processor Version Register

The Processor Version Register (PVR) is a 32-bit read-only register typically used to identify a specific processor core and chip implementation. Software can read the PVR to determine processor core and chip hardware features. The PVR can be read into a GPR using the **`mfspr`** instruction. Valid PVR values are `0x7FF21910`, `0x7FF21911`, and `0x7FF21912`.

Processor Identification Register

The Processor Identification Register (PIR) is a read-only register that uniquely identifies a specific instance of a processor core, enabling software to determine exactly which processor it is running on. The PIR can be read into a GPR using the **`mfspr`** instruction. Bits [0:27] of the PIR are reserved. The default value for bits [28:31] of the PIR is `1111`, but the value for these four bits can be defined by the user while configuring the processor as described in *PowerPC 440 Wrapper Data Sheet* [Ref 7].

Bit Settings for APU/FPU Usage

Whenever the APU/FPU is used, CCR0[9] must be set to `1`. Whenever the APU/FPU is used, CCR0[26] must be set to `1` in any operating environment where ITLB exceptions can occur. ITLB exceptions can occur in operating systems such as Linux. If the APU/FPU is not used, CCR0[26] must be set to `0` to avoid performance degradation.

Index